



# Creación de Aplicaciones Python

## Lección 6: API REST en Django

# ÍNDICE

<b>Lección 6: API REST en Django .....</b>	<b>1</b>
<b>Presentación y objetivos.....</b>	<b>1</b>
<b>1. ¿Qué es una API REST? .....</b>	<b>2</b>
<b>2. Códigos de estado de respuesta HTTP .....</b>	<b>3</b>
<b>3. API REST en Django .....</b>	<b>4</b>
<b>4. API REST usando Django REST Framework .....</b>	<b>11</b>
4.1 Método GET .....	12
4.2 Método POST.....	18
4.3 Método PUT .....	34
4.4 Método DELETE.....	43
<b>5. Puntos clave.....</b>	<b>49</b>

## Lección 6: API REST en Django

### PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos que es una API REST, cuáles son los métodos más empleados y como realizarlos usando Django.



#### **Objetivos**

- *¿Qué es API REST?*
- *Cuáles son los métodos de API REST y los códigos de estado.*
- *Crear una API REST usando Django y Django REST Framework.*

## 1. ¿QUÉ ES UNA API REST?

Antes de adentrarnos en el mundo REST , será necesario describir que es una interfaz es la conexión funcional entre dos sistemas, programas, o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles, permitiendo el intercambio de información.

Normalmente la interfaz tendremos una información de entrada que internamente la transforma o realiza una acción para dar lugar a información de salida.

Por lo tanto REST es una interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON.

Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: **POST** (crear), **GET** (leer y consultar), **PUT** (editar) y **DELETE** (eliminar).

Los objetos en REST siempre se manipulan a partir de la URI. Es la URI y ningún otro elemento el identificador único de cada recurso de ese sistema REST.

El formato más empleado en la actualidad es el formato JSON, ya que es más ligero y legible en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

Crea una petición HTTP que contiene toda la información necesaria, es decir, un REQUEST a un servidor tiene toda la información necesaria y solo espera una RESPONSE, ósea una respuesta en concreto.

### Ventajas:

- Nos permite separar el cliente del servidor. Esto quiere decir que nuestro servidor se puede desarrollar en Node y Express, y nuestra API REST con Django por ejemplo, no tiene por qué estar todos dentro de un mismo.
- Podemos crear un diseño de un microservicio orientado a un dominio (DDD)
- Es totalmente independiente de la plataforma, así que podemos hacer uso de REST tanto en Windows, Linux, Mac o el sistema operativo que nosotros queramos.
- Nos da escalabilidad, porque tenemos la separación de conceptos de CLIENTE y SERVIDOR, por tanto, podemos dedicarnos exclusivamente a la parte del servidor.

## 2. CÓDIGOS DE ESTADO DE RESPUESTA HTTP

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases:

1. Respuestas informativas (100–199),
2. Respuestas satisfactorias (200–299),
3. Redirecciones (300–399),
4. Errores de los clientes (400–499),
5. y errores de los servidores (500–599).

Vamos a centrarnos en las más empleadas:

### **200 OK**

La solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP:

### **201 Created**

La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT.

### **400 Bad Request**

Esta respuesta significa que el servidor no pudo interpretar la solicitud dada una sintaxis inválida.

### **401 Unauthorized**

Es necesario autenticar para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, la autenticación es posible.

### **404 Not Found**

El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.

### **500 Internal Server Error**

El servidor ha encontrado una situación que no sabe cómo manejarla.

### **501 Not Implemented (en-US)**

El método solicitado no está soportado por el servidor y no puede ser manejado. Los únicos métodos que los servidores requieren soporte (y por lo tanto no deben retornar este código) son GET y HEAD.

### 3. API REST EN DJANGO

La API REST en Django consta de la siguientes partes:

Back-end es la parte que se encuentra en servidores consta del propio proyecto Django más si queremos una base de datos. Esta se encargará de la gestión de los datos.

Front-end es la parte que interactúa con el usuario normalmente se usan plantillas HTML y CSS. Para su interacción con el proyecto Django se usan métodos HTTP como GET, POST, PUT y DELETE.

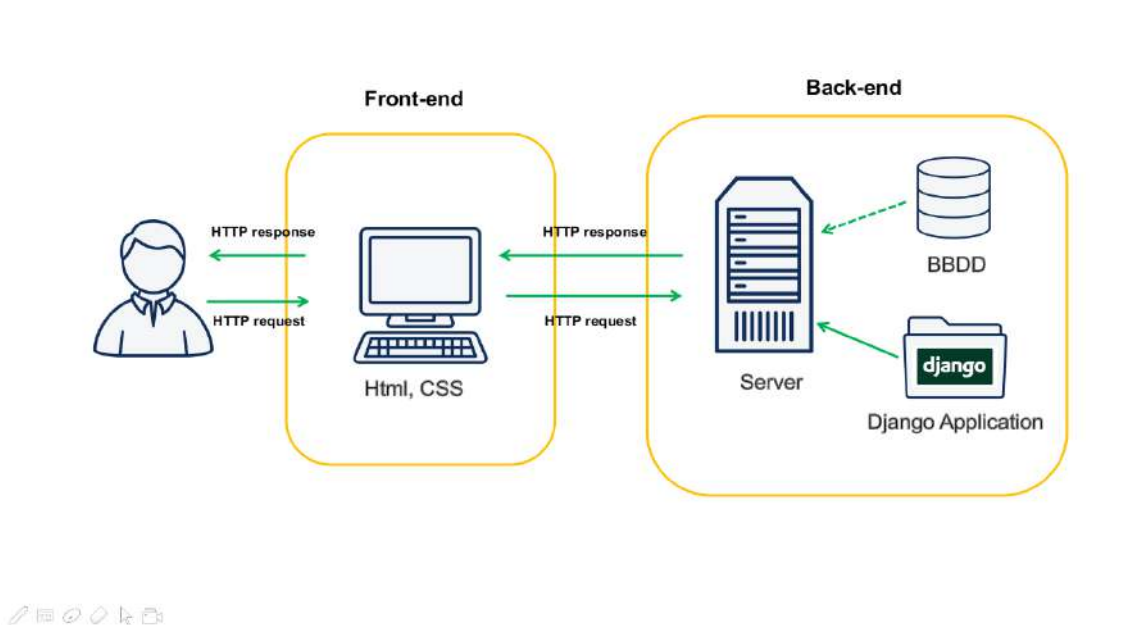


Figura 3.1: Arquitectura de las aplicaciones Django

En el caso de Django, como interaccionamos con el Front-end (HTML) solo será posible realizar métodos GET y POST, para realizar el resto de métodos para otros tipos de Front-end podemos chequearlos gracias a POSTMAN una herramienta que nos sirve de interacción con nuestra API REST.

Un ejemplo que vamos a usar es el registro de nuevos usuarios:

En myApp creamos un nuevo archivo form.py:

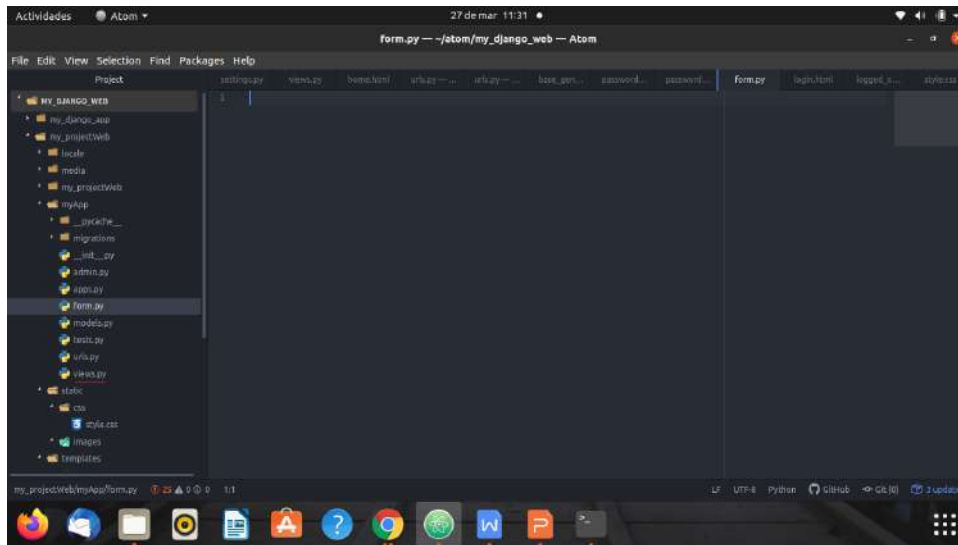


Figura 3.2: Archivo form.py

En el crearemos un formulario para el registro en el cual tenemos todos los datos que nos aporta django (user, password, etc) y añadimos el email, para ello ponemos lo siguiente:

```
from django.contrib.auth.forms import UserCreationForm

class CustomUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        fields = UserCreationForm.Meta.fields + ("email",)
```

Importamos la librería de django UserCreationForm y creamos la clase CustomUserCreationForm donde recogeremos los datos del usuario:

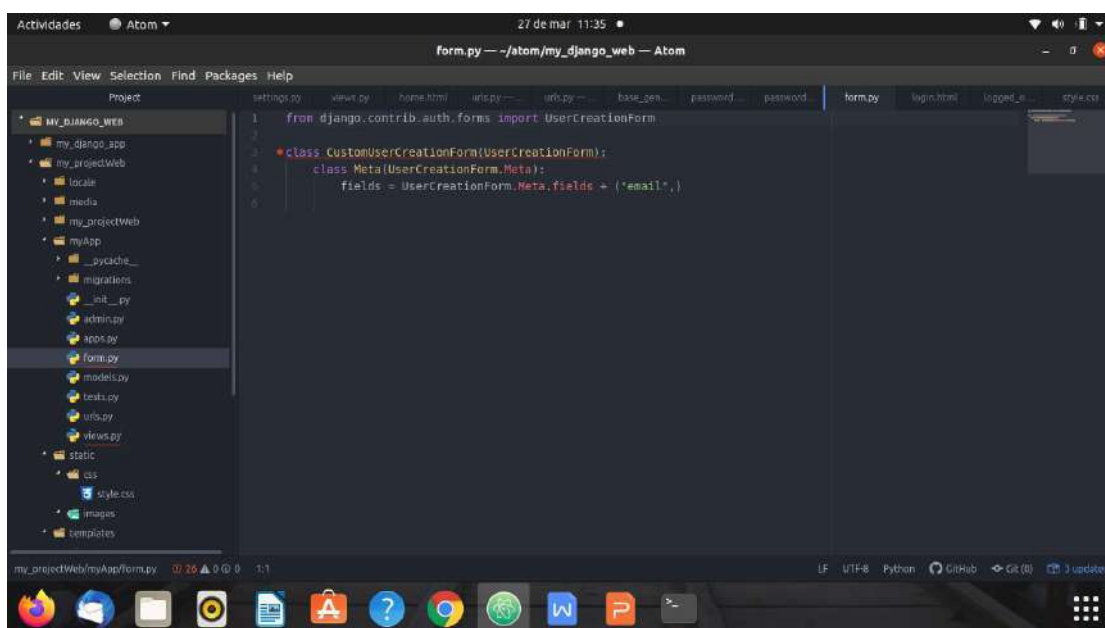


Figura 3.3: Archivo form.py

Donde tenemos nuestro archivo myApp --> views.py importamos las siguientes librerías:

#Login para volver a la función de entrada:

```
from django.contrib.auth import login
```

#Redirect, render y reverse para volver a la url principal (home):

```
from django.shortcuts import redirect, render  
from django.urls import reverse
```

#CustomUserCreationForm para llamar a la función antes creada:

```
from myApp.form import CustomUserCreationForm
```

Crearemos la función register para el registro de nuevos usuarios:

- 1) Un método GET donde cargaremos el formulario de registro para un nuevo usuario:

```
if request.method == "GET":  
    return render(  
        request, "users/register.html",  
        {"form": CustomUserCreationForm}  
    )
```

- 2) Un método POST: donde guardaremos los datos del usuario y volveremos a nuestra página home:

```
elif request.method == "POST":  
    form = CustomUserCreationForm(request.POST)  
    if form.is_valid():  
        user = form.save()  
        login(request, user)  
        return redirect(reverse("home"))
```



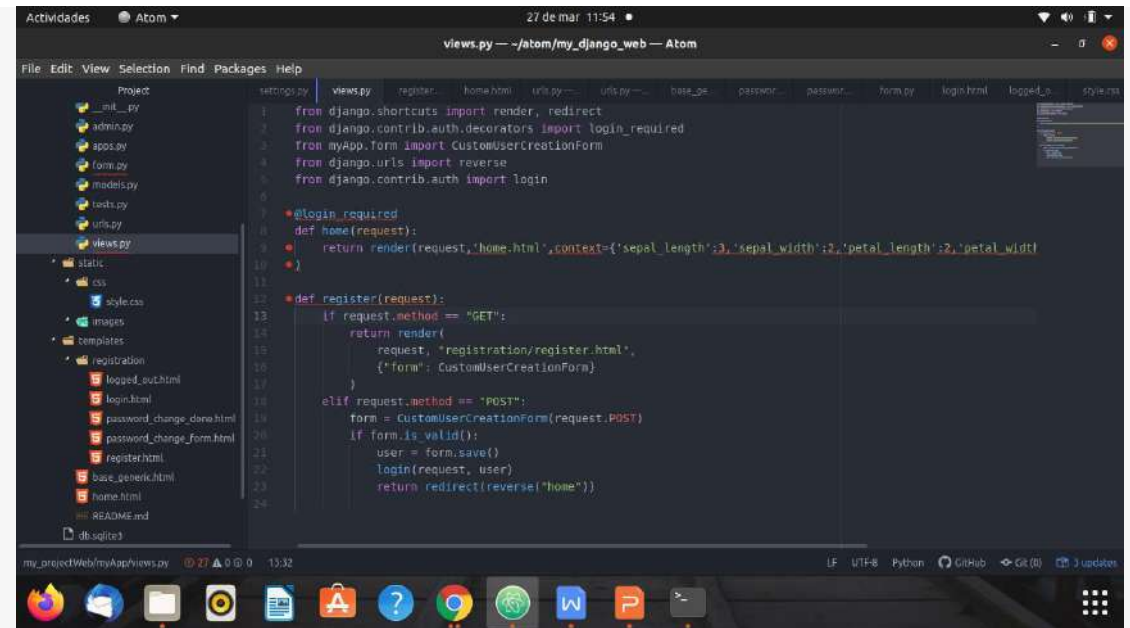


Figura 3.4: Archivo views.py

Creamos el html register.html donde describiremos el método a emplear `<form method="post">` y `{{form}}` que corresponde al cuestionario creado por django para el registro:

```
{% extends "base_generic.html" %}
{% block content %}
<h2>Register</h2>
<form method="post">
    {% csrf_token %}
    {{form}}
    <input type="submit" value="Register">
</form>
<a href="{% url 'login' %}">Back to login</a>
{% endblock %}
```

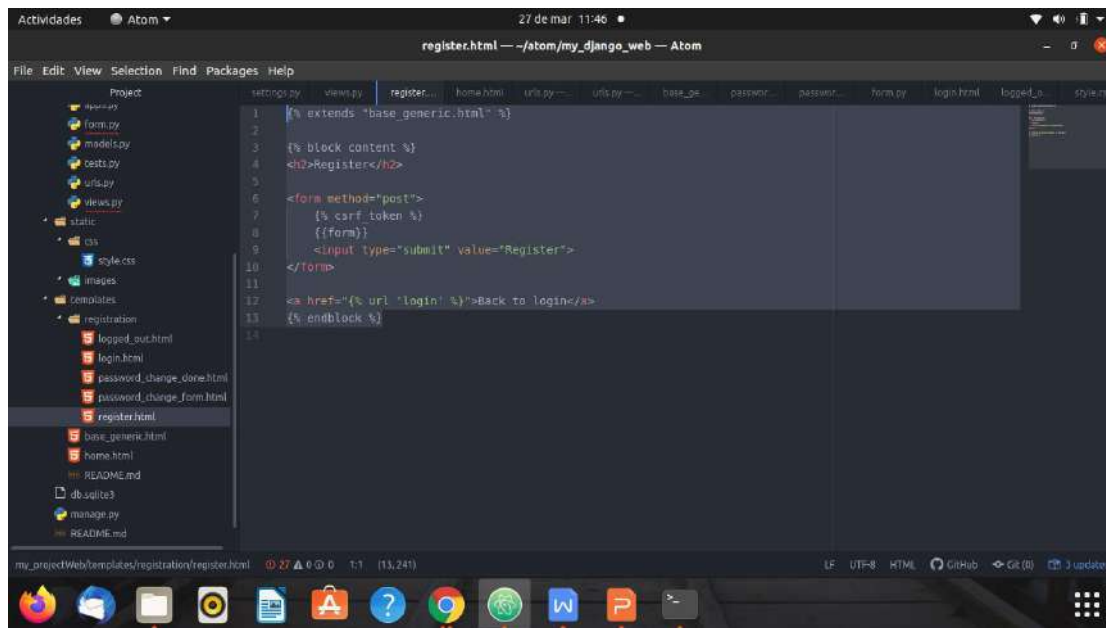


Figura 3.5: Archivo register.html

Añadimos la url al archivo myApp--> urls.py:

```
url(r"^register/", views.register, name="register"),
```

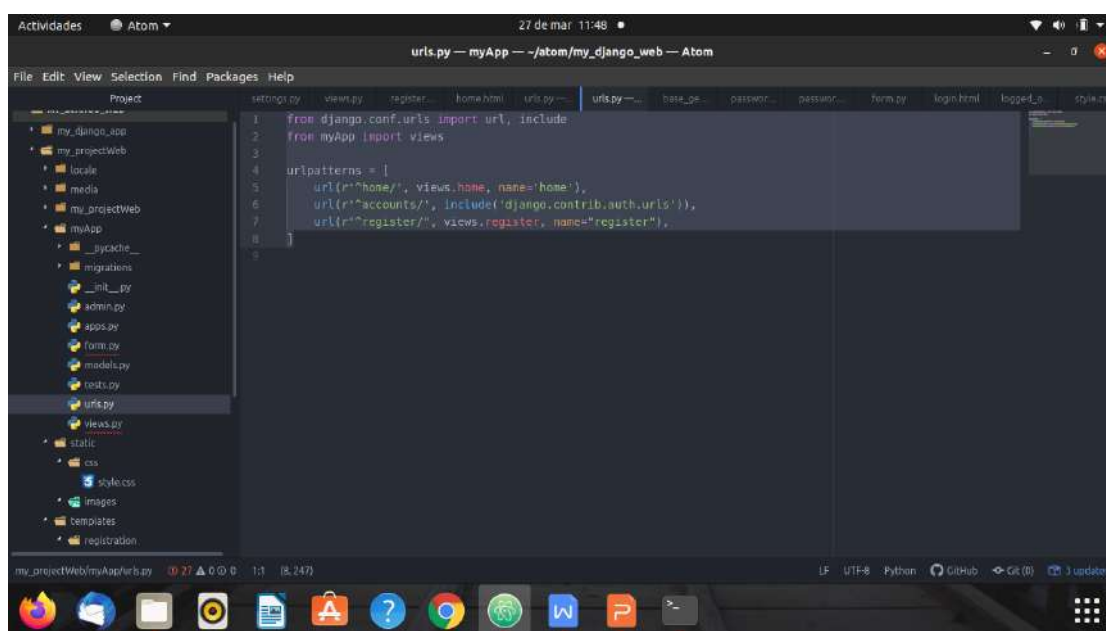


Figura 3.6: Archivo urls.py

Añadimos a login.html el botón para acceder a un nuevo registro mediante:

```
<p><a href="{% url 'register' %}">Register</a></p>
```

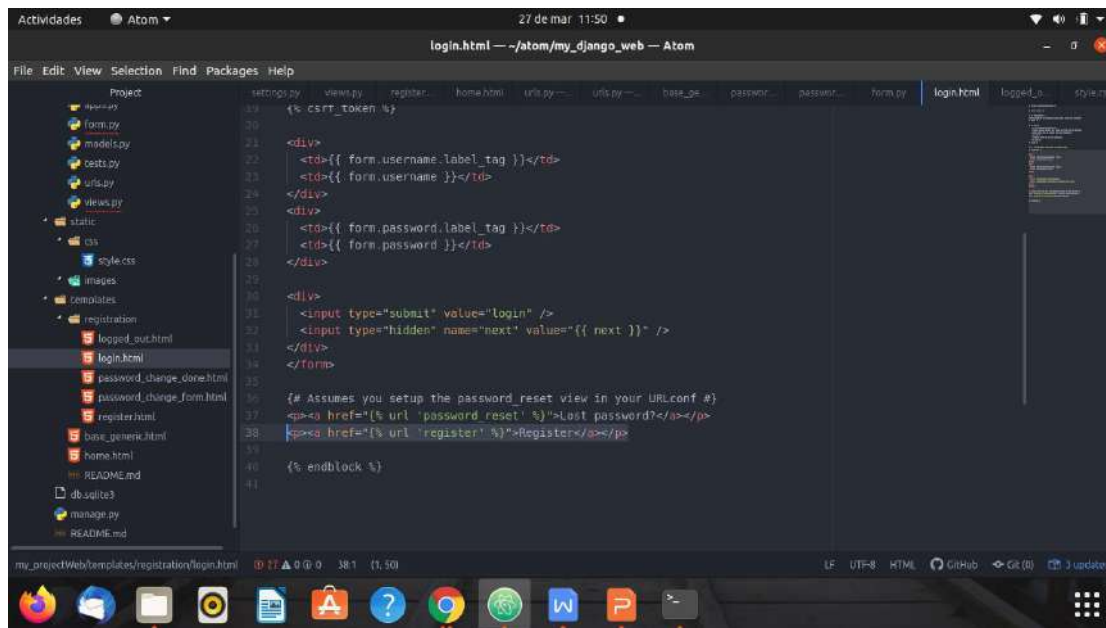


Figura 3.7: Archivo login.html

Para comprobarlo vamos a <http://127.0.0.1:8000/accounts/login/>

Vemos que se nos ha añadido un botón nuevo de register:

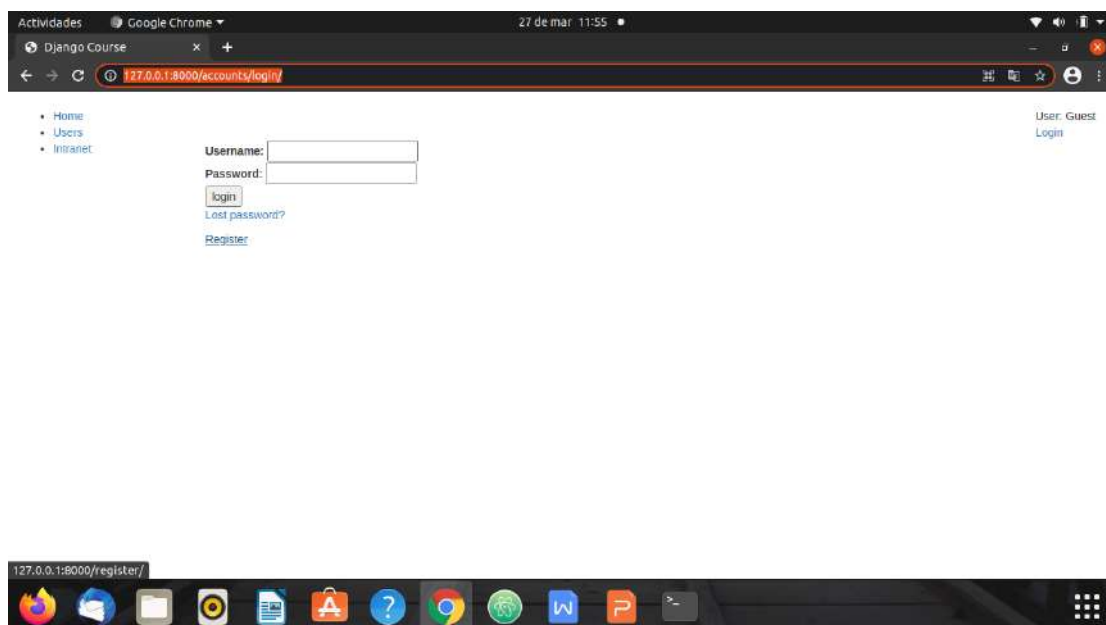
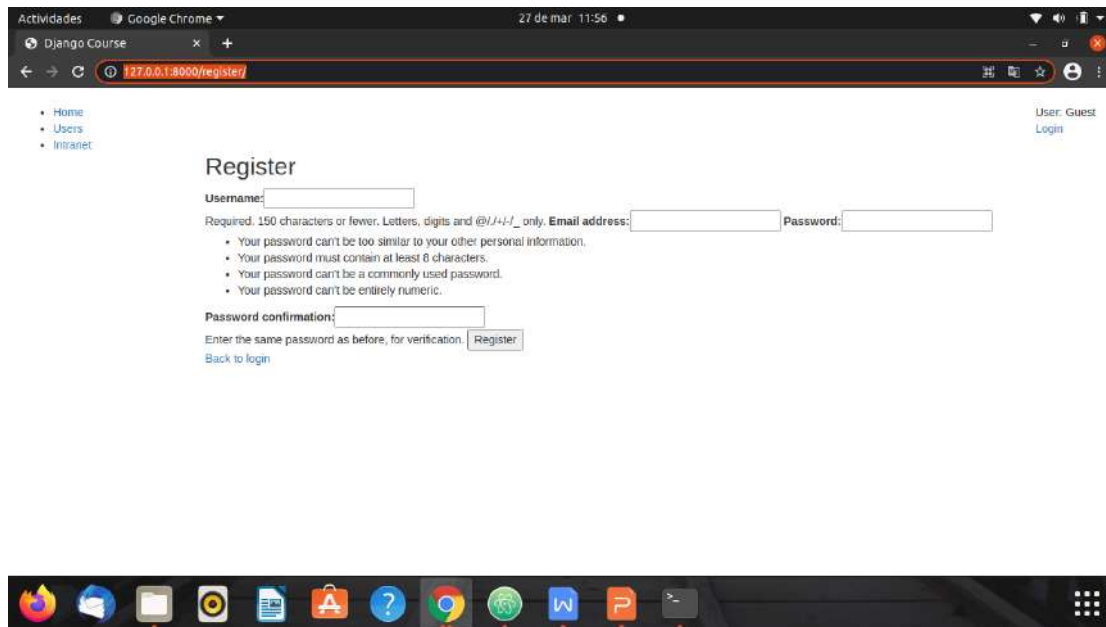


Figura 3.8: Pantalla de entrada añadido el botón Register

Pulsamos y nos abrirá un formulario como este:  
<http://127.0.0.1:8000/register/>



The screenshot shows a Google Chrome browser window with the address bar displaying `127.0.0.1:8000/register/`. The page title is 'Django Course'. The registration form includes fields for 'Username', 'Email address', and 'Password'. The password field has a list of requirements: 'Required: 150 characters or fewer. Letters, digits and @/./+/-/\_ only.', 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', 'Your password can't be a commonly used password.', and 'Your password can't be entirely numeric.'. There is also a 'Password confirmation' field with the instruction 'Enter the same password as before, for verification.' and a 'Register' button. A 'Back to login' link is at the bottom left. The top right shows 'User: Guest' and a 'Login' link. The bottom of the screen shows a Linux desktop taskbar with various application icons.

Figura 3.9: Pantalla de Registrar un nuevo usuario

## 4. API REST USANDO DJANGO REST FRAMEWORK

Es uno de los framework más empleados en Django gracias a que tiene librerías para los códigos de estados (status), respuestas (Response), métodos a emplear (Api View), para mejor uso de los modelos de datos por el serializer, etc y nos permite ver los errores que surgen.

Para ello en my\_projectWeb --> settings.py añadimos:

```
INSTALLED_APPS = [ ...,  
    'rest_framework',  
    'myApp',]
```

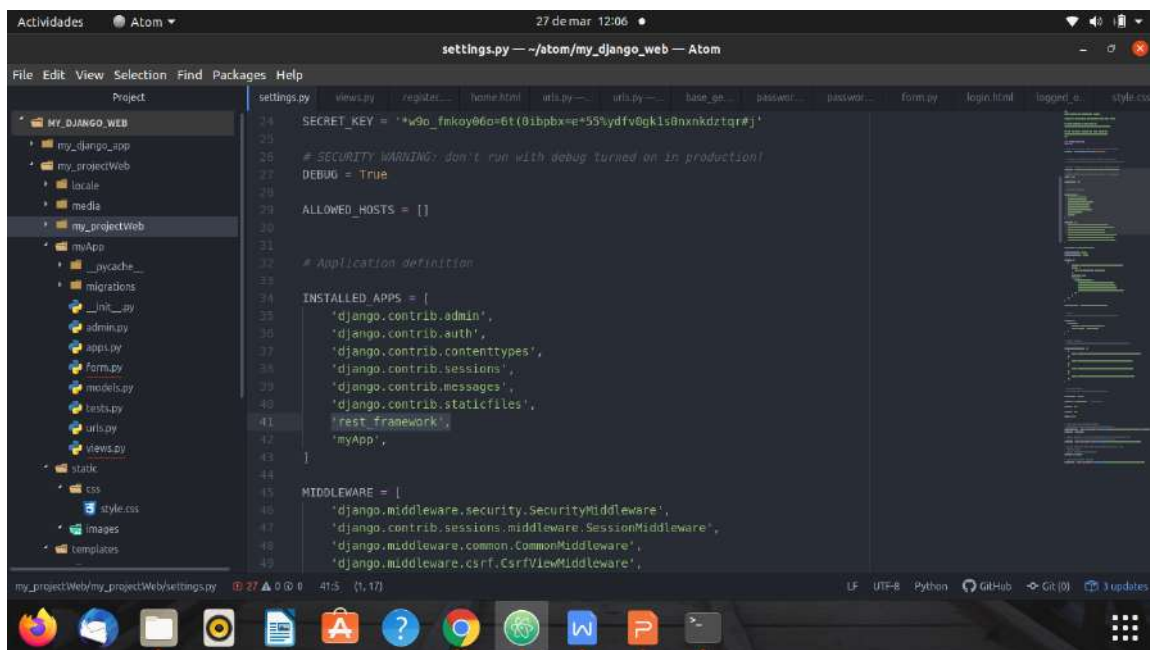


Figura 4.1: Archivo settings.py

Será necesario instalar pandas para nuestro entorno virtual:

```
pip install pandas
```

```

Actividades Terminal 27 de mar 12:52
Isabel@Isabel-SVE1512E1EW: ~/atom/my_django_web/my_projectWeb

(my_django_app) Isabel@Isabel-SVE1512E1EW:~/atom/my_django_web/my_projectWeb$ pip install pandas
Collecting pandas
  Using cached pandas-1.2.3-cp38-cp38-manylinux1_x86_64.whl (9.7 MB)
Requirement already satisfied: pytz>=2017.3 in /home/isabel/atom/my_django_web/my_django_app/lib/python3.8/site-packages (from pandas) (2021.1)
Requirement already satisfied: python-dateutil>=2.7.3 in /home/isabel/atom/my_django_web/my_django_app/lib/python3.8/site-packages (from pandas) (2.8.1)
Requirement already satisfied: numpy>=1.16.5 in /home/isabel/atom/my_django_web/my_django_app/lib/python3.8/site-packages (from pandas) (1.20.1)
Requirement already satisfied: six>=1.5 in /home/isabel/atom/my_django_web/my_django_app/lib/python3.8/site-packages (from python-dateutil>=2.7.3->pandas) (1.14.0)
Installing collected packages: pandas
Successfully installed pandas-1.2.3
(my_django_app) Isabel@Isabel-SVE1512E1EW:~/atom/my_django_web/my_projectWeb$
  
```

Figura 4.2: Instalación pandas

## 4.1 Método GET

Vamos a coger el dataset de Iris para realizar nuestras pruebas para ello colocamos el dataset en nuestra carpeta media --> iris.csv:

```

Actividades Atom 27 de mar 15:18
iris.csv --atom/my_django_web -- Atom

File Edit View Selection Find Packages Help

Project settings.py views.py iris.csv main.html

* my_django_web
  * my_django_app
  * my_projectWeb
    * locale
    * media
    * iris.csv
    * my_projectWeb
    * myApp
    * __pycache__
    * __pycache__
    * views.py
    * migrations
    * __init__.py
    * admin.py
    * aspx.py
    * form.py
    * models.py
    * test.py
    * urls.py
    * views.py
    * static

1 sepal length, sepal width, petal length, petal width, species
2 5.1,3.5,1.4,0.2,setosa
3 4.9,3.0,1.4,0.2,setosa
4 4.7,3.2,1.3,0.2,setosa
5 4.6,3.1,1.5,0.2,setosa
6 5.0,3.6,1.4,0.2,setosa
7 5.4,3.9,1.7,0.4,setosa
8 4.6,3.4,1.4,0.3,setosa
9 5.0,3.4,1.5,0.2,setosa
10 4.4,2.9,1.4,0.2,setosa
11 4.9,3.1,1.5,0.1,setosa
12 5.4,3.7,1.5,0.2,setosa
13 4.8,3.4,1.6,0.2,setosa
14 4.8,3.0,1.4,0.1,setosa
15 4.3,3.0,1.1,0.1,setosa
16 5.0,4.0,1.2,0.2,setosa
17 5.7,4.4,1.5,0.4,setosa
18 5.4,3.9,1.3,0.4,setosa
19 5.1,3.5,1.4,0.3,setosa
20 5.7,3.8,1.7,0.3,setosa
21 5.1,3.8,1.5,0.3,setosa
22 5.4,3.4,1.7,0.2,setosa
23 5.1,3.7,1.5,0.4,setosa
24 4.6,3.6,1.0,0.2,setosa
25 5.1,3.3,1.7,0.5,setosa
26 4.8,3.4,1.9,0.2,setosa
  
```

Figura 4.3: Archivo Iris.csv

Vamos a crear nuestra plantilla para la aplicación creamos una carpeta en templates--> iris --> main.html:

```
{% extends "base_generic.html" %}
{% block content %}
<html>
  <body>
    <h1>Iris Dataset</h1>
    <br>
    <h3>• Resume:</h3>
    <table class="table table-hover">
      <tr>
        <td></td>
        <td><label>Sepal Length</label></td>
        <td><label>Sepal Width</label></td>
        <td><label>Petal Length</label></td>
        <td><label>Petal Width</label></td>
      </tr>
      {% for key, value in describe.items %}
        <tr>
          <td><strong>{{ key }}</strong></td>
          <td>{{ value.sepal_length }}</td>
          <td>{{ value.sepal_width }}</td>
          <td>{{ value.petal_length }}</td>
          <td>{{ value.petal_width }}</td>
        </tr>
      {% endfor %}
    </table>
    <br>
    <h3>• All Data:</h3>
    <table class="table table-hover">
      <tr>
        <td></td>
        <td><label>Sepal Length</label></td>
        <td><label>Sepal Width</label></td>
        <td><label>Petal Length</label></td>
        <td><label>Petal Width</label></td>
        <td><label>Species</label></td>
      </tr>
      {% for key, value in data.items %}
        <tr>
          <td><strong>{{ key }}</strong></td>
          <td>{{ value.sepal_length }}</td>
          <td>{{ value.sepal_width }}</td>
          <td>{{ value.petal_length }}</td>
          <td>{{ value.petal_width }}</td>
          <td>{{ value.species }}</td>
        </tr>
      {% endfor %}
    </table>
  </body>
</html>
{% endblock %}
```



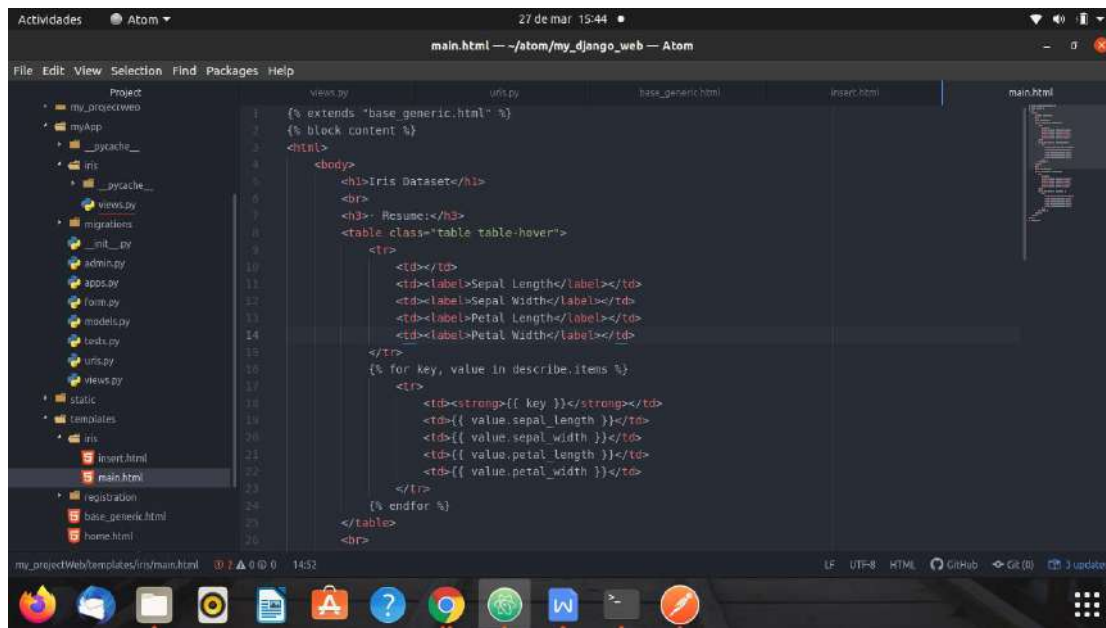


Figura 4.4: Archivo main.html

Una vez creado vamos a crear nuestra función de Python creamos una carpeta en nuestra aplicación myApp--> iris-->views.py:

Importamos las siguientes librerías:

```
from rest_framework.decorators import api_view
from rest_framework import status
```

Crearemos los métodos permitidos para esta función gracias al decorador api\_view de rest\_framework y daremos el status code para la respuesta.

Cargamos la librería responsable de mostrar los datos y la plantilla que vamos a usar:

```
from django.shortcuts import render
```

Daremos la ruta donde se encuentra nuestro archivo que está descrito en las settings.py:

```
from django.conf import settings
```

Librerías pandas y json para cargar los datos y mostrarlos:

```
import pandas as pd
import json
```

Cuando carguemos la página nos mostrará un resume de nuestros datos y el dataset completo, con ayuda del método GET:

```
@api_view(['GET'])
def irisData(request):
```



```

if request.method == 'GET':

    # Ruta donde se encuentra nuestro archivo:

    # /home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv

    X = settings.MEDIA_ROOT + '/iris.csv'

    # Cargamos el dataset con ayuda de pandas:

    X_df = pd.read_csv(X)

    # Lo transformamos a json para poder gestionarlo desde el html:

    data = X_df.to_json(orient="index")

    data = json.loads(data)

    # Resumen del Dataset .describe():

    describe = X_df.describe().to_json(orient="index")

    describe = json.loads(describe)

    # Context son los datos que pasamos al html

    return render(request, 'iris/main.html', context={'data': data,
'describe': describe}, status=status.HTTP_200_OK)

```

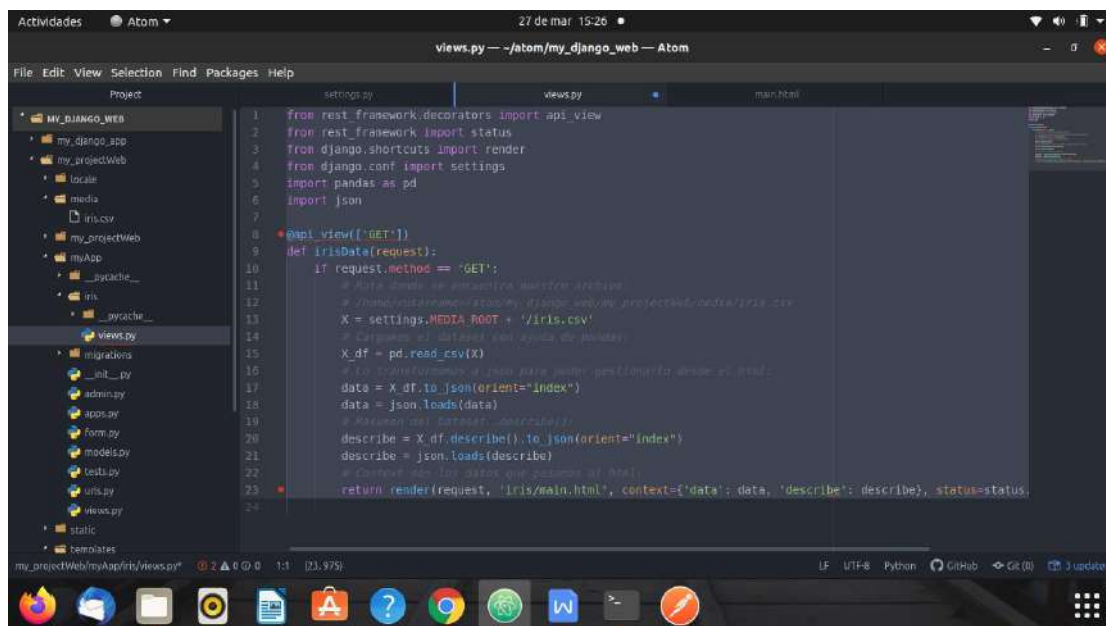


Figura 4.5: Archivo views.py

Una vez que tenemos las vistas y la función la nombramos en el archivo myApp-->urls.py:

```

# ruta donde esta nuestra función:

from myApp.iris.views import irisData

urlpatterns = [...,

    url(r"^iris/", irisData, name="iris"),

]

```

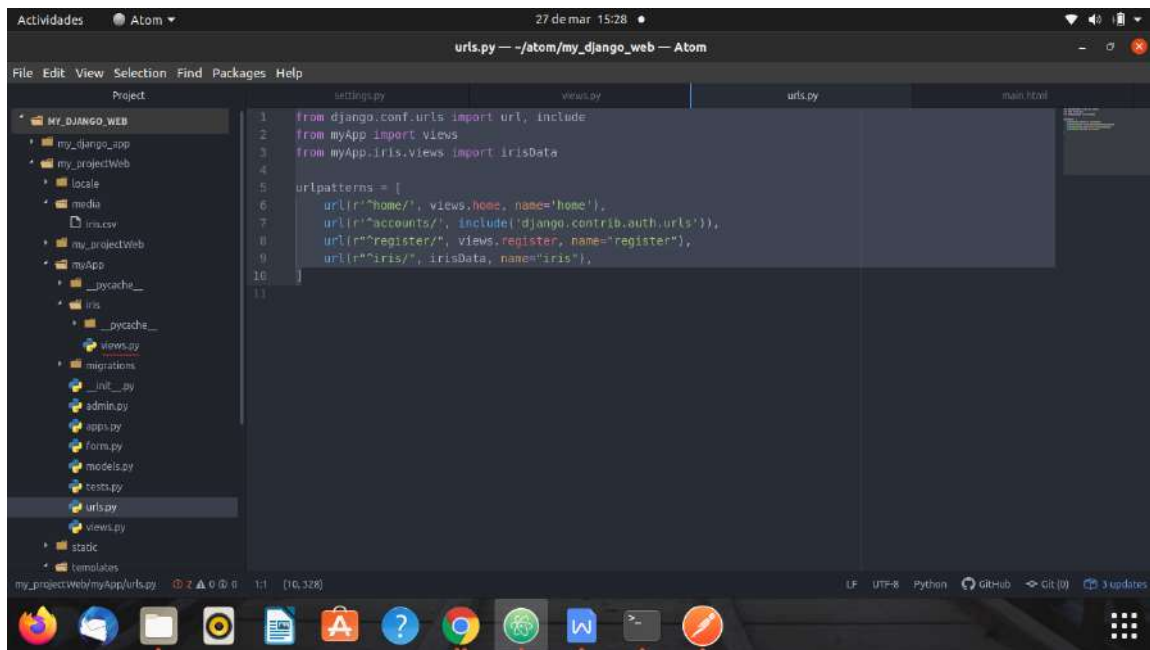


Figura 4.6: Archivo urls.py de la aplicación

Para poder acceder a esta página que hemos programado necesitamos llamar a la url en el menú de la izquierda vamos a templates--> base\_generic.html añadimos:

```
<li><a href="{% url 'iris' %}">Iris Data</a></li>
```

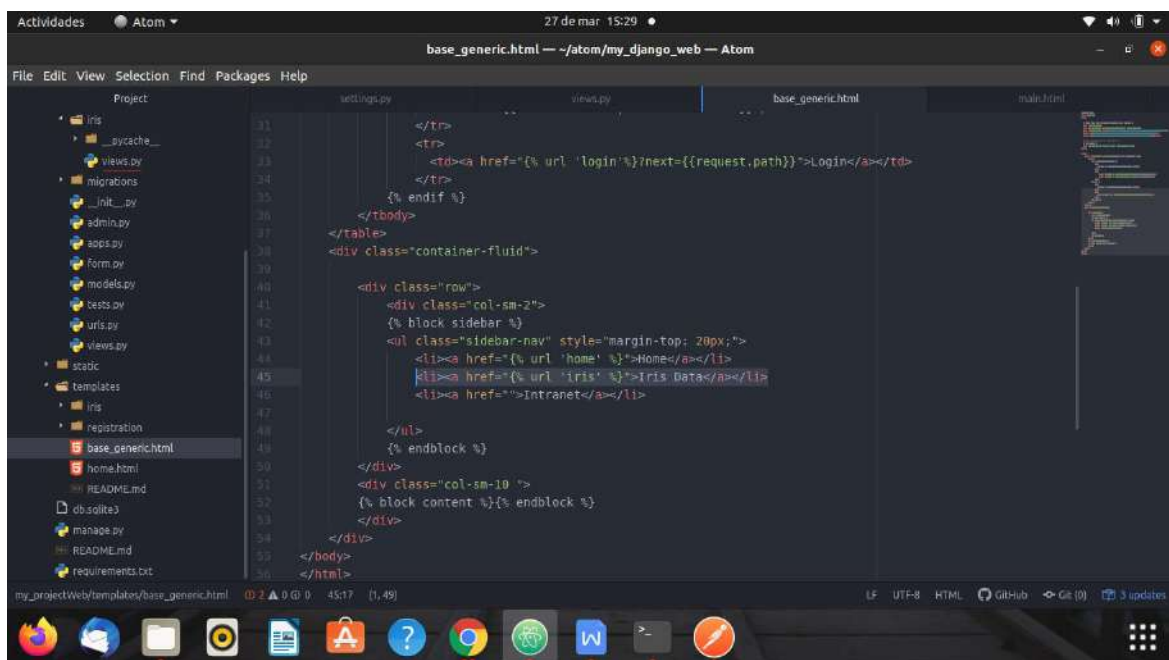


Figura 4.7: Archivo urls.py de la aplicación

Vemos que en el menú se nos muestra **Iris Data**:

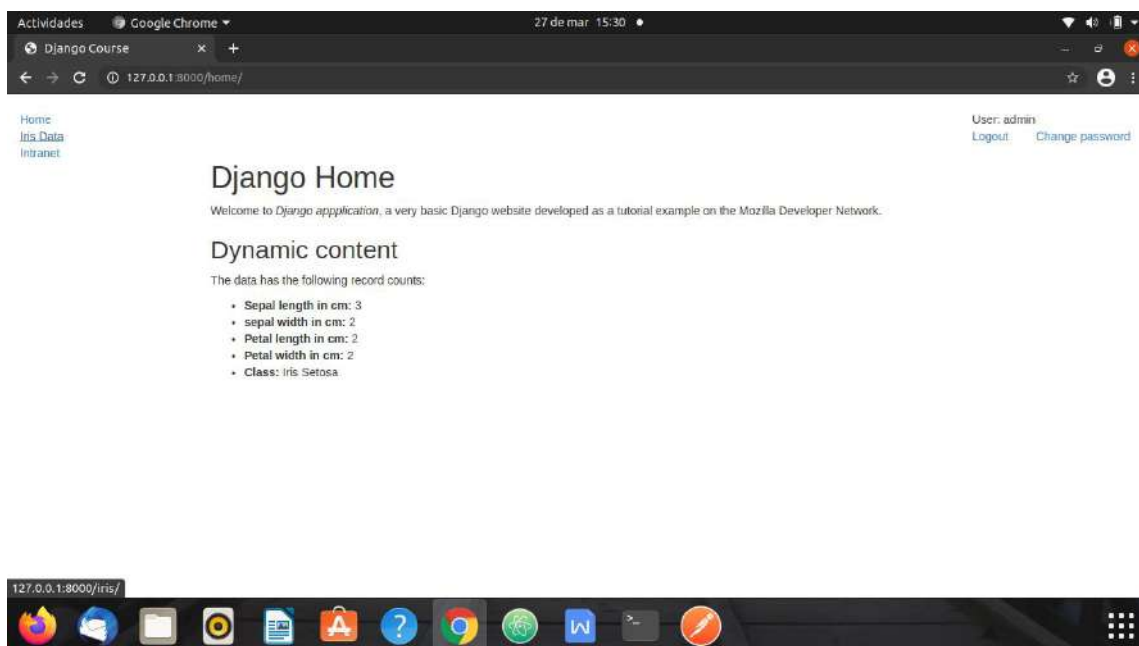


Figura 4.8: Pantalla de Inicio donde se muestra el botón Iris Data en el menú de la izquierda arriba.

Hacemos clic se nos muestra la siguiente imagen:

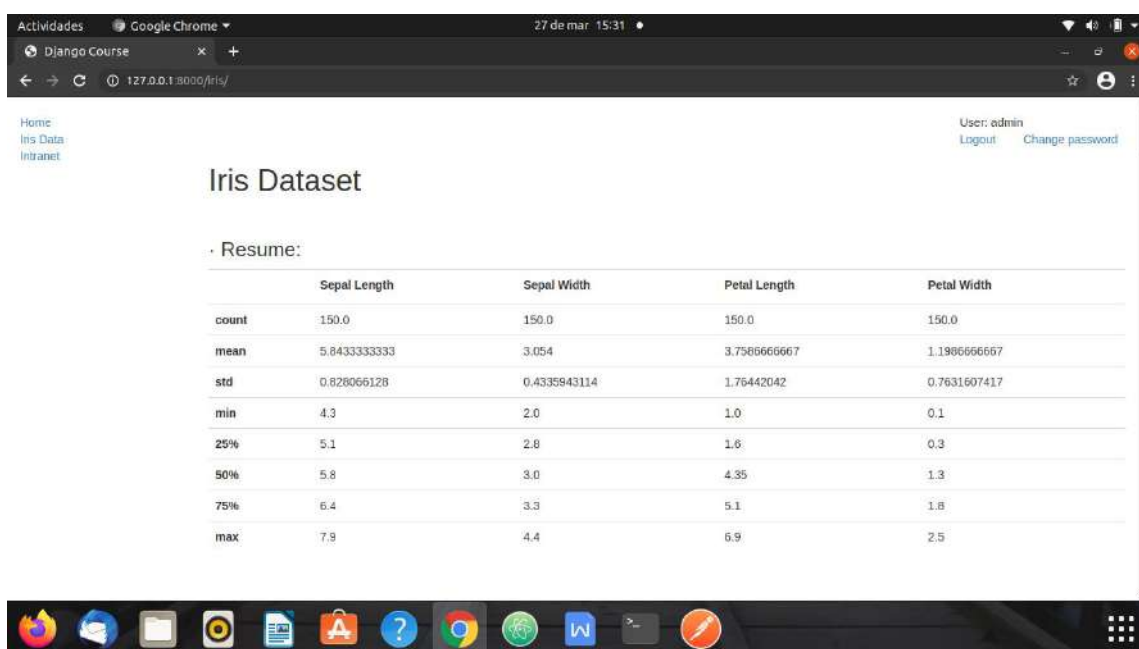


Figura 4.9: Pantalla de Iris Data.

Para obtener la misma plantilla usando Postman copiamos la url:  
<http://127.0.0.1:8000/iris/>

La ponemos en Postman donde pone GET damos a **Send**:

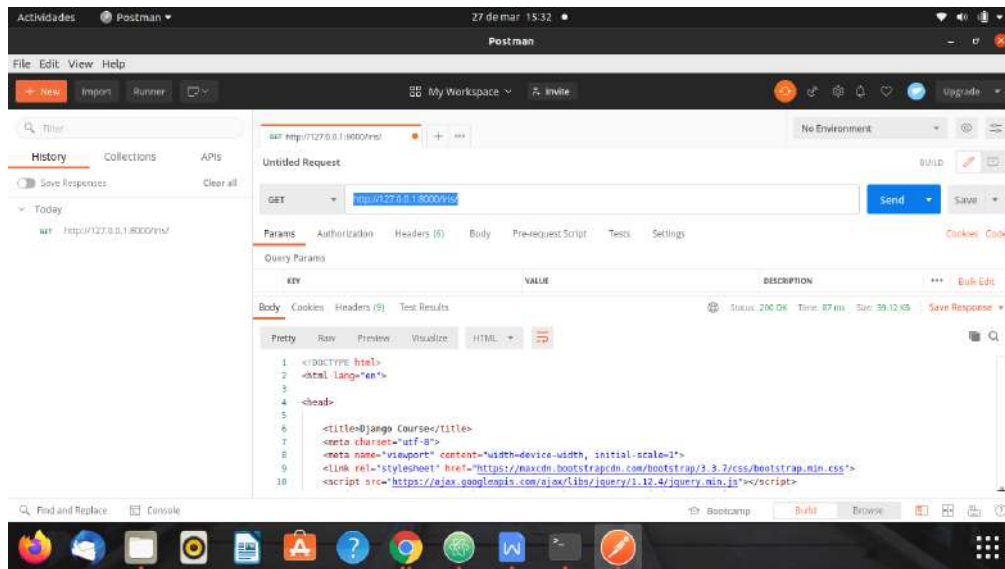


Figura 4.10: Vista de Postman GET a <http://127.0.0.1:8000/iris/>

Vemos que nos ha dado un 200 OK en color verde y nos muestra el resultado de la plantilla de HTML que hemos usado.

## 4.2 Método POST

Creamos la función POST para este método:

```
@api_view(['GET', 'POST'])
def insertData(request):
    return render(request, 'iris/isert.html')
```

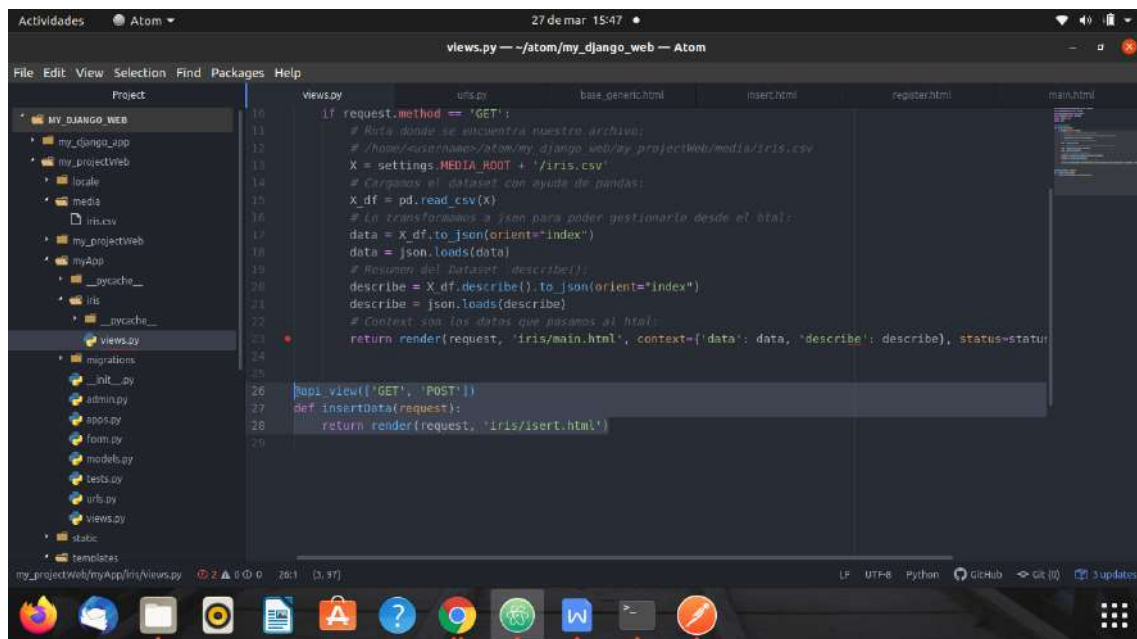


Figura 4.11: Archivo views.py añadido método POST

Nombramos la url donde nos vamos dirigir:

```
from myApp.iris.views import irisData, insertData
```

```
urlpatterns = [ ..., url(r'^insertData/', insertData, name="insertData"),]
```

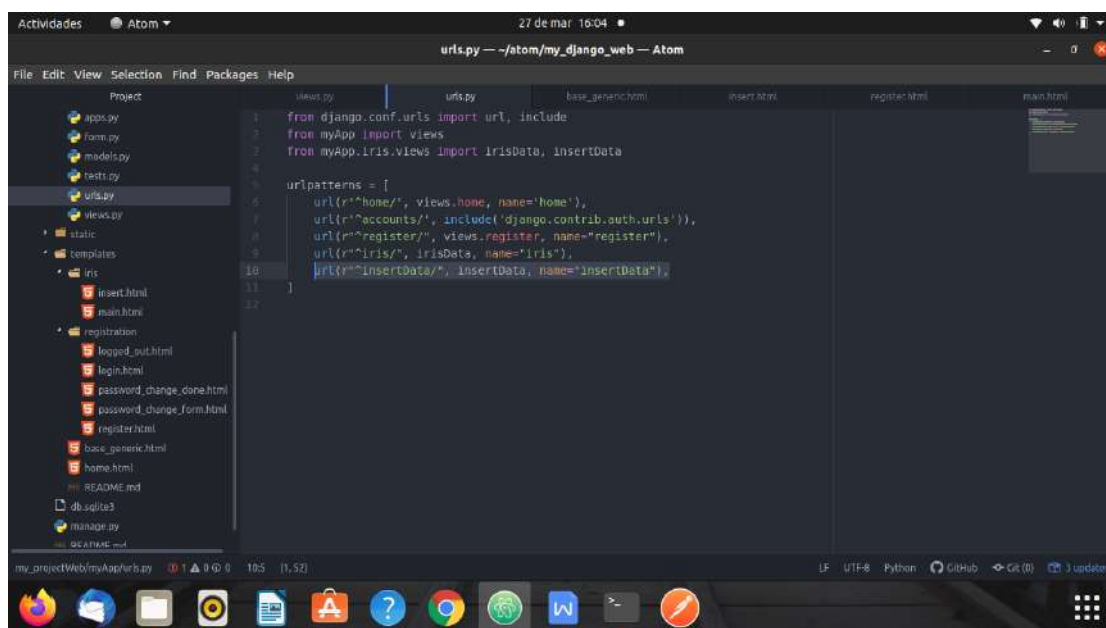


Figura 4.12: Archivo urls.py de la aplicación.

Lo nombramos en el Menú de la izquierda para poder acceder a ella:

```
<li><a href="{% url 'insertData' %}">Insert Data</a></li>
```

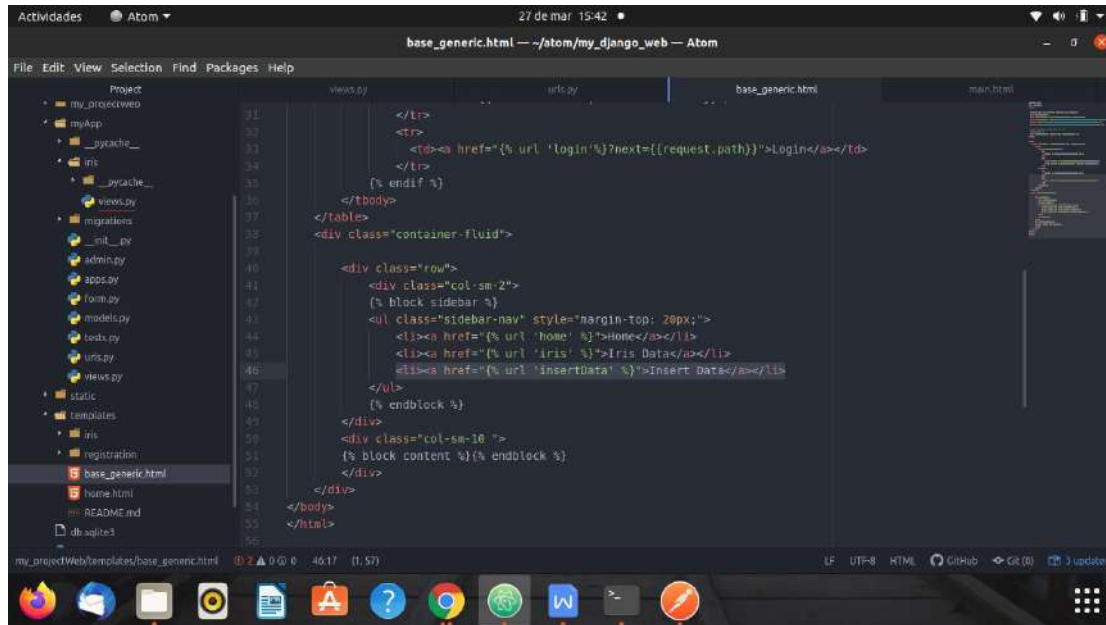


Figura 4.13: Archivo basic\_generic.py añadiendo el botón Insert data

Realizamos la plantilla para poder insertar los datos templates-->iris-->insert.html:

```
{% extends "base_generic.html" %}
{% block content %}
<html>
  <body>
    <form method='post'>
      {% csrf_token %}
      <h1>Iris Dataset</h1>
      <br>
      <h3>· Insert Data:</h3>
      {% if result != '' %}
        <label style="color:green;">{{ result }}</label>
      {% endif %}
      <table class="table table-hover">
        <tr>
          <th><label>Sepal Length:</label></th>
          <td><input
            type="number" step='0.1' /></td>
          <td><input type="text" value="" /></td>
        </tr>
```



```

        <tr>
            <th><label>Sepal Width:</label></th>
            <td><input          name="sepal_width"          value=""
type="number" step='0.1' /></td>
        </tr>
        <tr>
            <th><label>Petal Length:</label></th>
            <td><input          name="petal_length"         value=""
type="number" step='0.1' /></td>
        </tr>
        <tr>
            <th><label>Petal Width:</label></th>
            <td><input          name="petal_width"          value=""
type="number" step='0.1' /></td>
        </tr>
        <tr>
            <th><label>Species:</label></th>
            <td><input          name="species"              value=""
type="text" /></td>
        </tr>
    </table>
    <input type="submit" name="Submit">
</form>
</body>
</html>
{% endblock %}

```

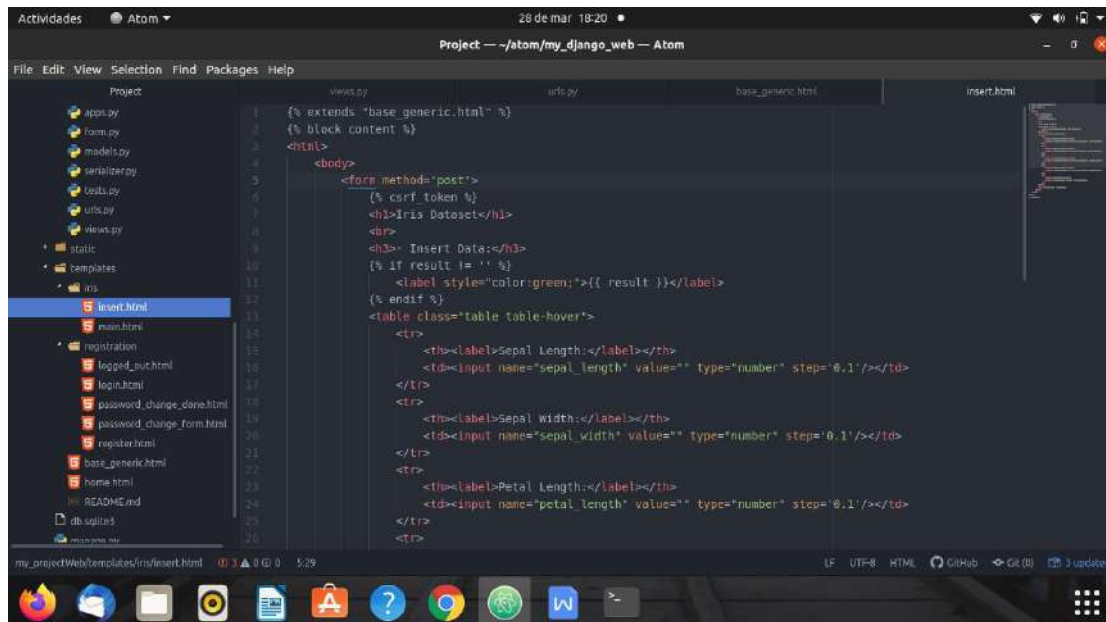


Figura 4.14: Archivo inset.html

Si vamos a nuestra aplicación y pulsamos en el menu Insert Data:

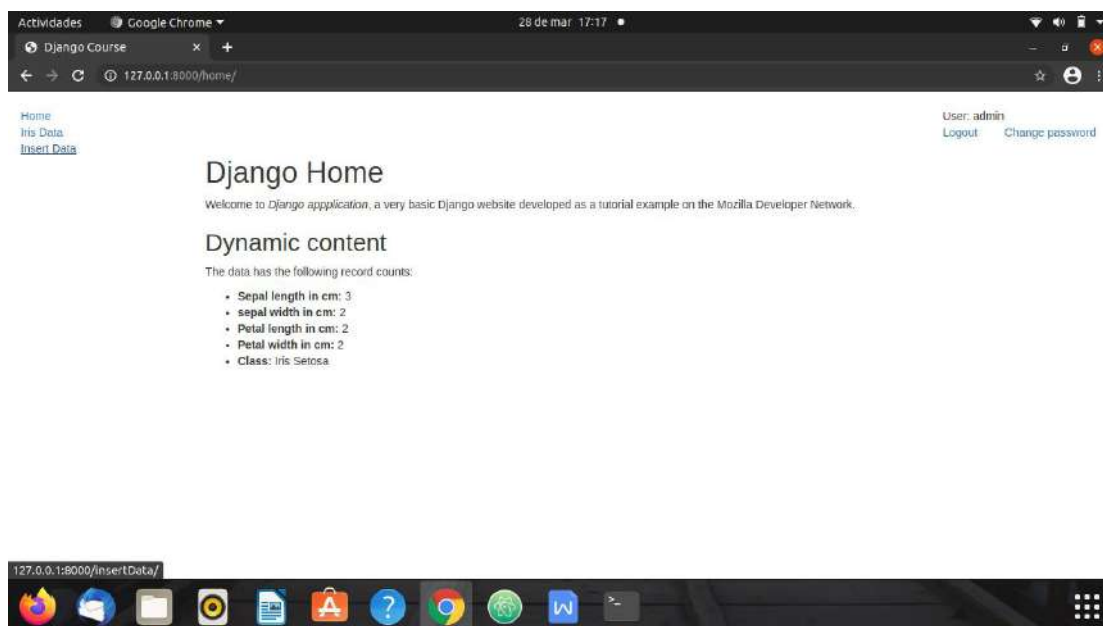


Figura 4.15: Pantalla de Inicio donde aparece el botón Insert Data

Nos mostrará la plantilla que acabamos de programar:

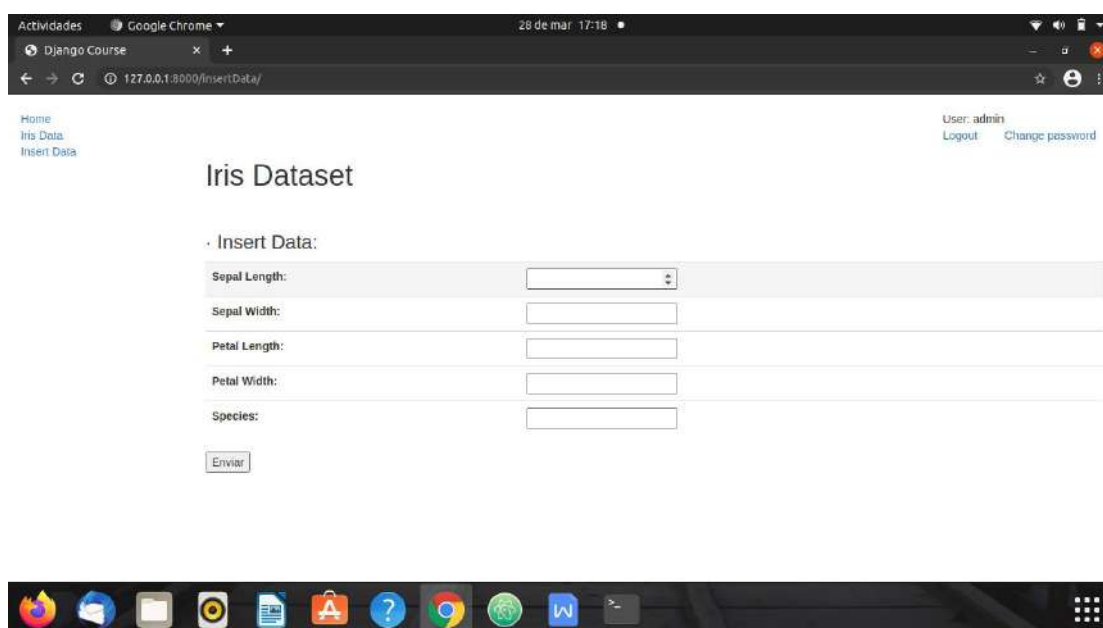


Figura 4.16: Pantalla para insertar datos en el csv



## Modelo de datos:

Vamos a crear nuestro modelo de datos para cuando recibamos la información esté en el formato correcto, definiendo que tipo de variable se espera, si son decimales (float), si es un texto (char), si es un numero entero (integer), etc. Podemos encontrarnos:

### *A. Argumentos comunes de los campos*

Los siguientes argumentos son comunes a la mayoría de los tipos de campo y pueden usarse al declararlos:

- **default:** Valor por defecto para el campo.
- **null:** Si es True, Django guardará valores en blanco o vacíos como NULL en la base de datos para campos donde sea apropiado (un CharField guardará una cadena vacía en su lugar). Por defecto es False.
- **blank:** Si es True, se permite que el campo quede en blanco en tus formularios. El valor por defecto es False, lo que significa que la validación de formularios de Django te forzará a introducir un valor. Con frecuencia se usa con null=True, porque si vas a permitir valores en blanco, también querrás que la base de datos sea capaz de representarlos de forma apropiada.
- **primary\_key:** Si es True, establece el campo actual como clave primaria para el modelo (Una clave primaria es una columna especial de la base de datos, diseñada para identificar de forma única todos los diferentes registros de una tabla). Si no se especifica ningún campo como clave primaria, Django añadirá automáticamente un campo para este propósito.

### *B. Tipos comunes de campos*

La lista siguiente describe algunos de los tipos de campo más comúnmente usados.

- **CharField** se usa para definir cadenas de longitud corta a media. Debes especificar la max\_length (longitud máxima) de los datos que se guardarán.
- **TextField** se usa para cadenas de longitud grande o arbitraria. Puedes especificar una max\_length para el campo, pero sólo se usa cuando el

campo se muestra en formularios (no se fuerza al nivel de la base de datos).

- **IntegerField** es un campo para almacenar valores de números enteros y para validar los valores introducidos como enteros en los formularios.
- **DateField** y **DateTimeField** se usan para guardar/representar fechas e información fecha/hora (como en los objetos Python `datetime.date` y `datetime.datetime`, respectivamente). Estos campos pueden adicionalmente declarar los parámetros (mutuamente excluyentes) `auto_now=True` (para establecer el campo a la fecha actual cada vez que se guarda el modelo), `auto_now_add` (para establecer sólo la fecha cuando se crea el modelo por primera vez), y `default` (para establecer una fecha por defecto que puede ser sobrescrita por el usuario).
- **FileField** e **ImageField** se usan para subir ficheros e imágenes respectivamente (el `ImageField` añade simplemente una validación adicional de que el fichero subido es una imagen). Éstos tienen parámetros para definir cómo y donde se guardan los ficheros subidos.
- **AutoField** es un tipo especial de `IntegerField` que se incrementa automáticamente. Cuando no especificas una clave primaria para tu modelo, se añade -automáticamente- una de este tipo.

Para ello vamos a nuestra carpeta `myApp` y tenemos un archivo llamado `models.py` donde nombraremos nuestro modelo de datos que esperamos en la clase `IrisModel`:

```
from django.db import models
# Create your models here.
class irisModel(models.Model):
    sepal_length = models.FloatField(blank=True, default=0)
    sepal_width = models.FloatField(blank=True, default=0)
    petal_length = models.FloatField(blank=True, default=0)
    petal_width = models.FloatField(blank=True, default=0)
    species = models.CharField(max_length=30, blank=True, default='')

```

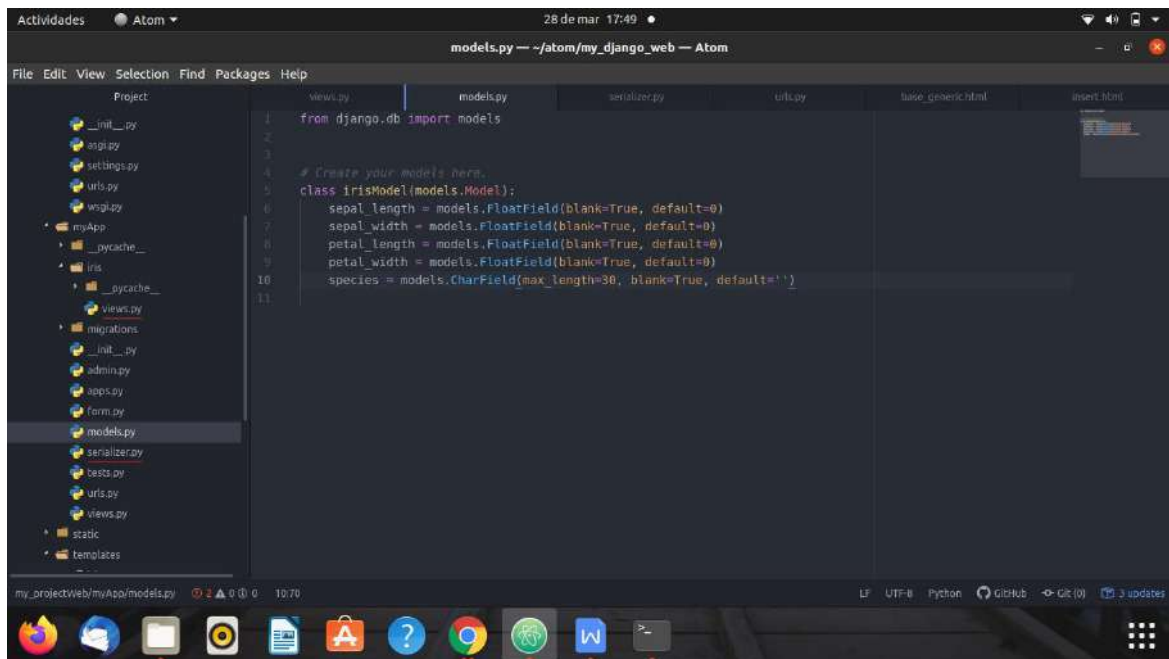


Figura 4.17: Archivo models.py

A continuación creamos el archivo serializer.py en nuestra app (myApp):

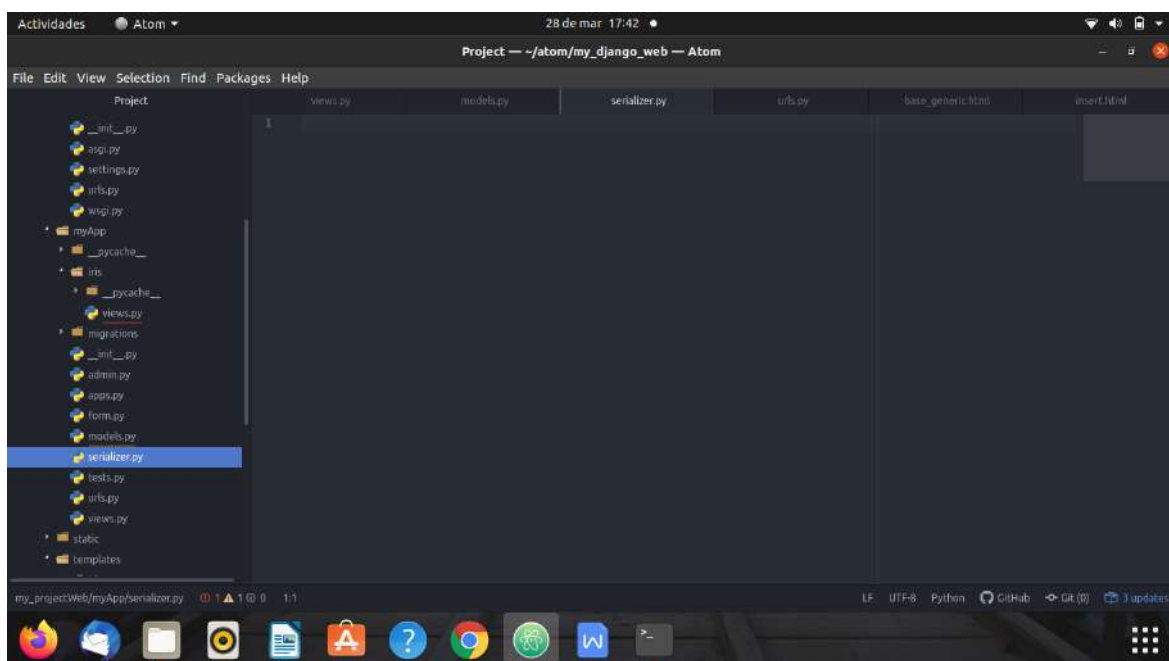


Figura 4.18: Archivo serializer.py

En él crearemos una función que recorra nuestro modelo de datos y compruebe que está en formato correcto.

Creamos la clase Meta donde definimos el model y en fields ponemos que nos tenga en cuenta todos los campos:

```
from rest_framework import serializers
from myApp.models import irisModel

class irisSerializer(serializers.ModelSerializer):
    class Meta:
        model = irisModel
        fields = '__all__'
```

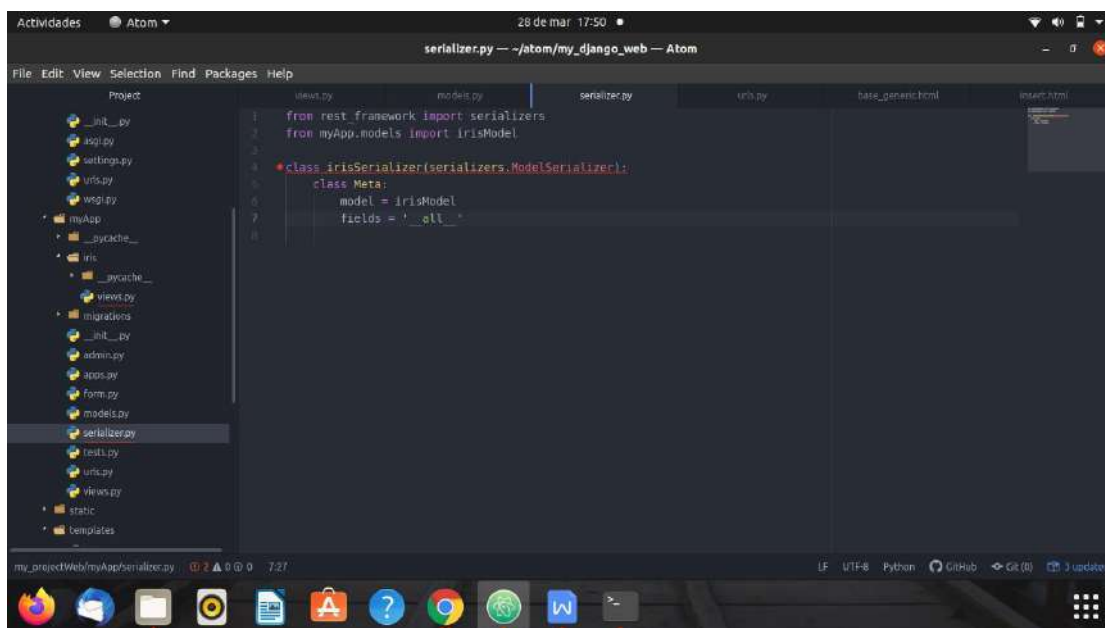


Figura 4.19: Archivo serializer.py

Volvemos a nuestro método Post (myApp--> iris-->views.py):

```
from myApp.serializer import irisSerializer
from django.shortcuts import render
from rest_framework.response import Response
import csv
...
@api_view(['GET', 'POST'])
def insertData(request):
    result = ''
    if request.method == 'GET':
        return render(request, 'iris/insert.html')
    elif request.method == 'POST':
        # definir 'data' como el conjunto de datos
        # que se reciben a través del front-end:
        data = request.data
        print(data)
        # Pasaremos los datos por el serializer
        # para comprobar si los datos siguen el modelo esperado:
        serializer = irisSerializer(data=data)
        if serializer.is_valid():
            # Guardamos los datos:
            serializer.save()
            # insertar dato en csv:
            X = settings.MEDIA_ROOT + '/iris.csv'
            with open(X, 'a', newline='') as csvfile:
                fieldnames = ['sepal_length', 'sepal_width',
                              'petal_length', 'petal_width', 'species']
                writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
                writer.writerow({'sepal_length': data['sepal_length'],
                                'sepal_width': data['sepal_width'],
                                'petal_length': data['petal_length'],
                                'petal_width': data['petal_width'],
                                'species': data['species']})

            print("writing complete")
            # resultado que nos muestre si se ha insertado:
            result = 'Insertado correctamente'
            return render(request, 'iris/insert.html',
                          context={'result': result}, status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                          status=status.HTTP_400_BAD_REQUEST)
```

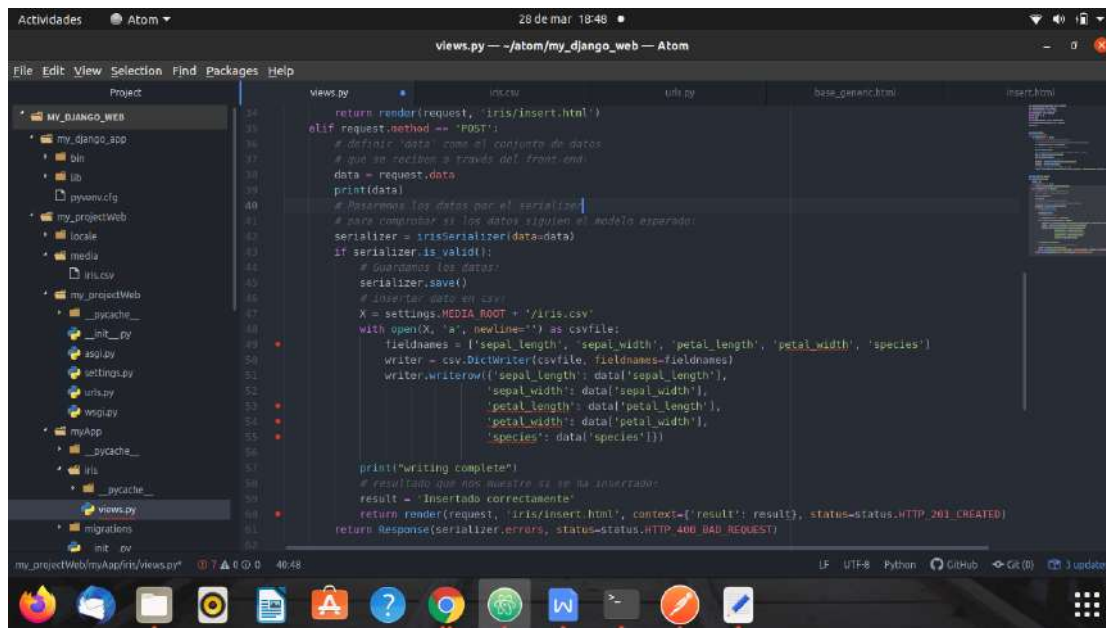


Figura 4.20: Archivo views.py

Una vez que hemos finalizado necesitamos migrar el modelo para que la aplicación nos lo tenga en cuenta para ello observamos que en la carpeta myApp--> migrations no existe ningún modelo aún migrado, necesitamos ir a la consola y ejecutar los siguientes comandos:

```
python manage.py makemigrations <name aplicación>
```

```
python manage.py migrate
```

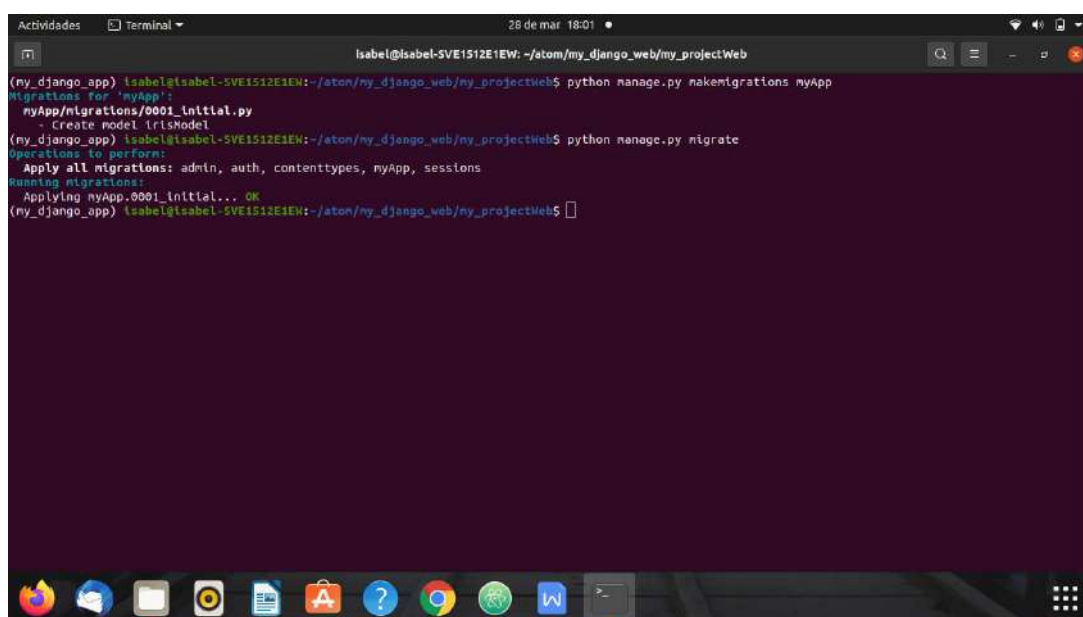


Figura 4.21: Migración del modelo en cmd

Observamos que se ha migrado correctamente pone un OK y vemos que se nos ha creado en la carpeta myApp --> migrations --> 0001\_initial.py donde aparece nuestro modelo de datos:

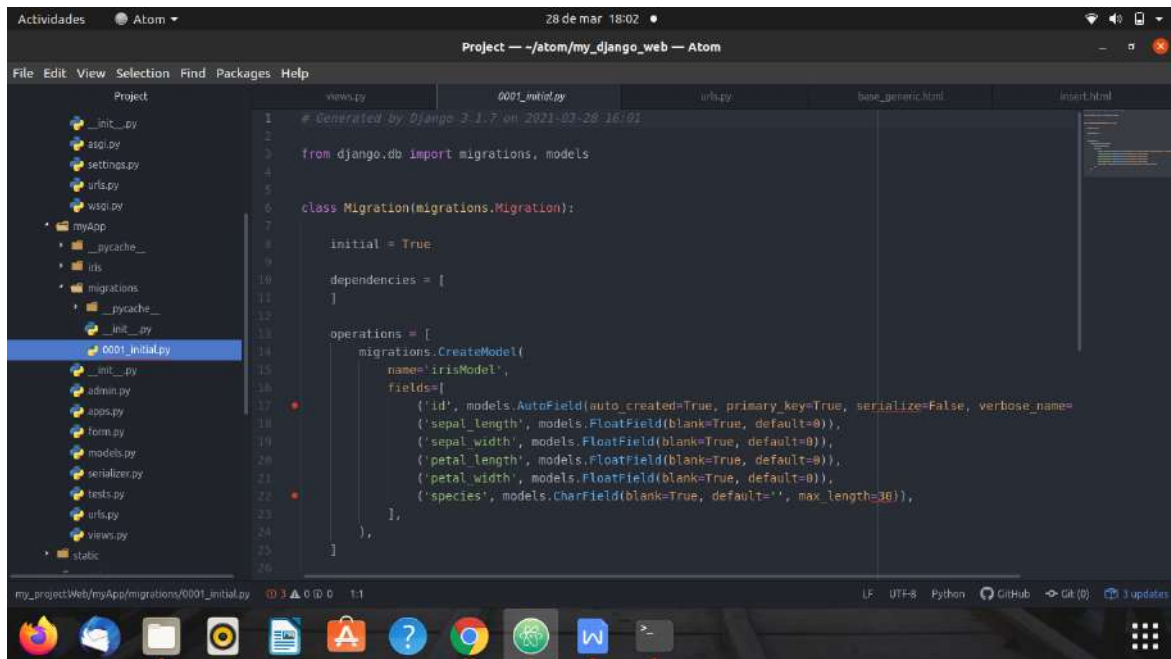


Figura 4.22: Archivo 0001\_initial.py

Cada vez que modifiquemos o creamos un modelo nuevo será necesario eliminar este archivo y volver a realizar los comando anteriores para actualizar la aplicación.

Volveremos a ejecutar nuestra aplicación con el comando:

```
Python manage.py runserver
```

Probamos a insertar un dato a través de nuestro front-end:  
<http://127.0.0.1:8000/insertData/>

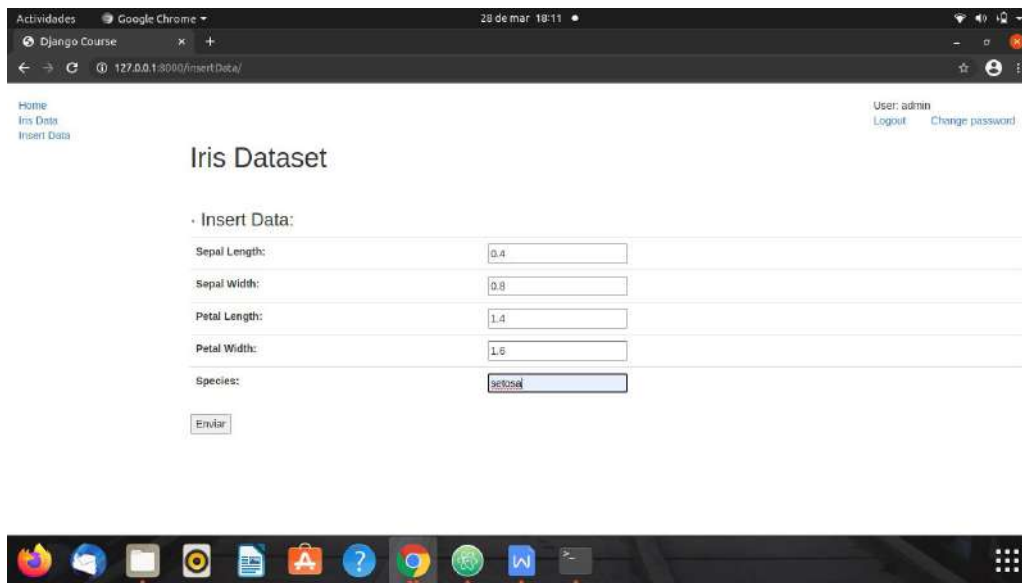


Figura 4.23: Prueba en el navegador de inserción de un dato.

Damos a enviar y vemos que si es correcto, nos mostrará el mensaje en verde de que se ha insertado correctamente:

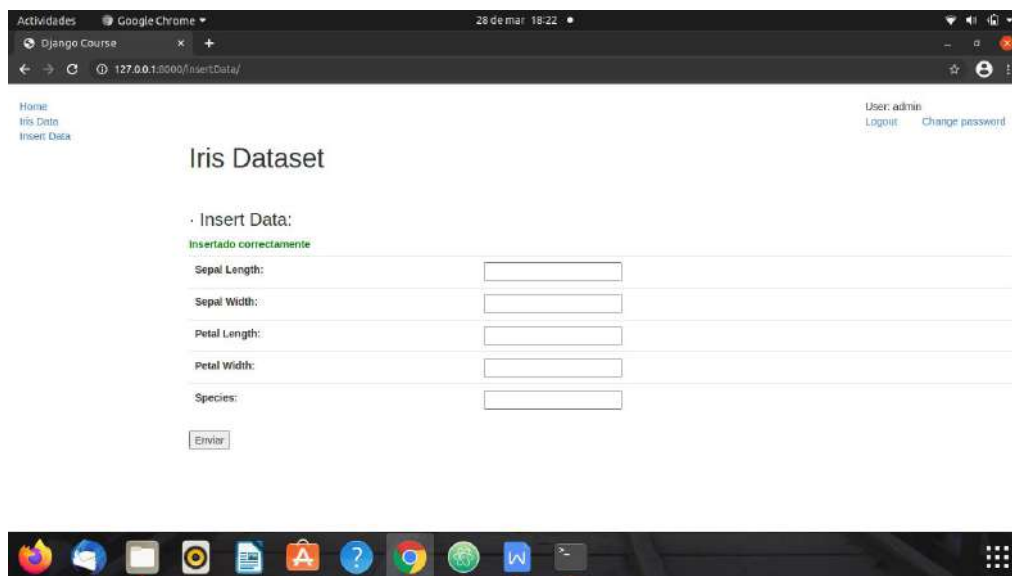
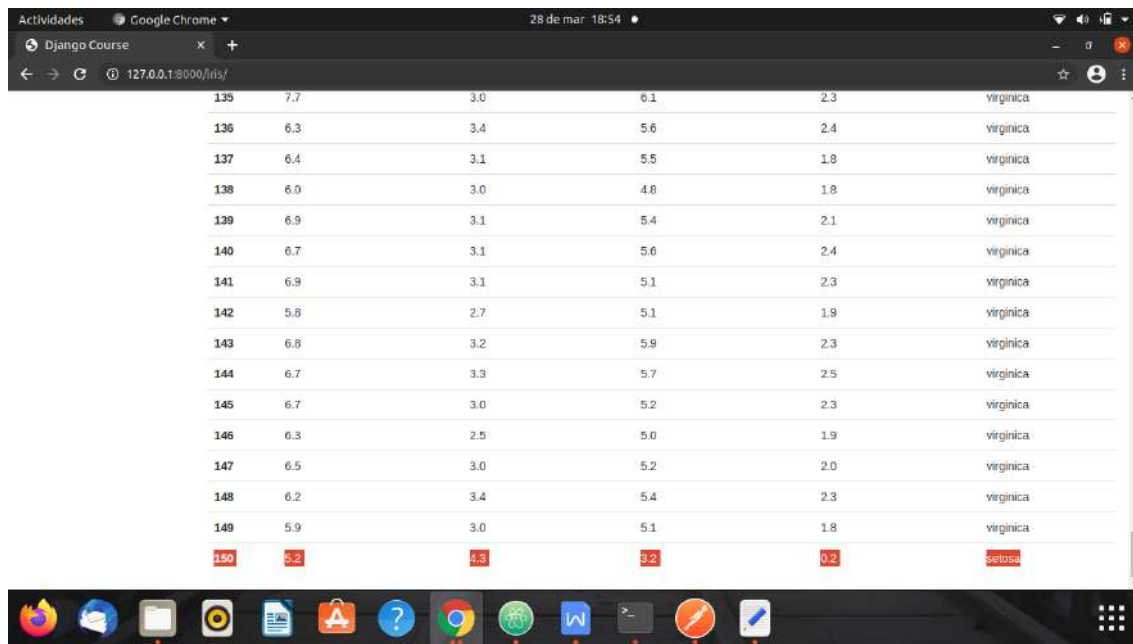


Figura 4.24: Prueba en el navegador de deserción de un dato que se ha insertado correctamente.

Si vamos a <http://127.0.0.1:8000/iris/> veremos que se nos ha agregado un dato al final del archivo.





135	7.7	3.0	6.1	2.3	virginica
136	6.3	3.4	5.6	2.4	virginica
137	6.4	3.1	5.5	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica
139	6.9	3.1	5.4	2.1	virginica
140	6.7	3.1	5.6	2.4	virginica
141	6.9	3.1	5.1	2.3	virginica
142	5.8	2.7	5.1	1.9	virginica
143	6.8	3.2	5.9	2.3	virginica
144	6.7	3.3	5.7	2.5	virginica
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica
150	3.2	4.3	3.2	0.2	setosa

Figura 4.25: Comprobación de deserción de un dato

Si vamos al POSTMAN:

Seleccionamos POST ponemos la url: <http://127.0.0.1:8000/insertData/>

Seleccionamos BODY--> raw--> json:

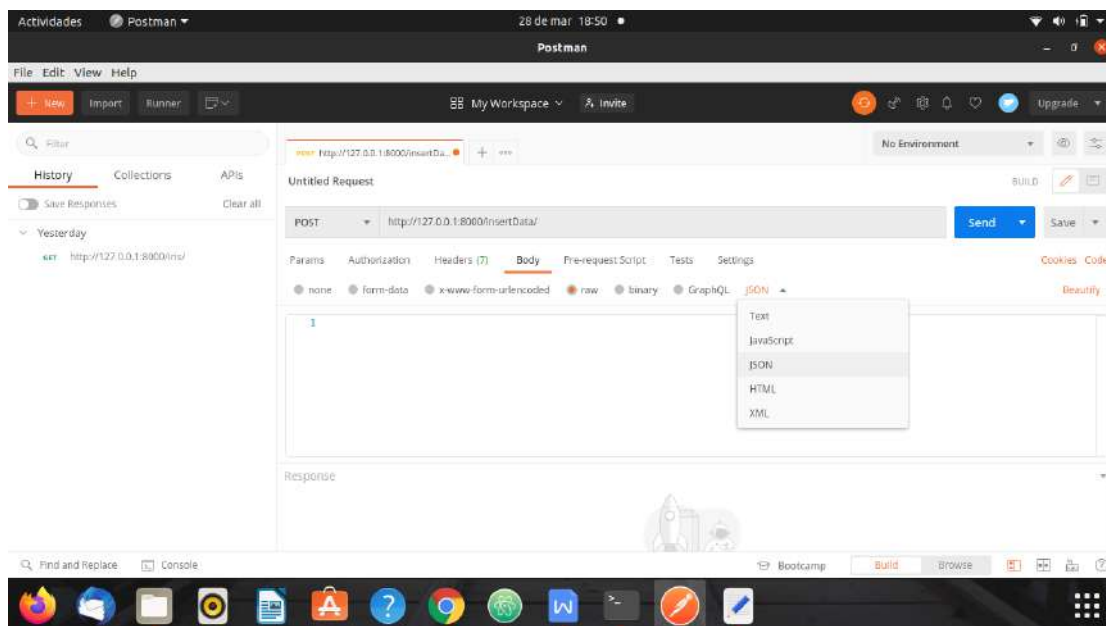


Figura 4.26: Comprobación de deserción de un dato en Postman

Ponemos:

```
{
  "sepal_length": 5.2,
  "sepal_width": 4.3,
  "petal_length": 3.2,
  "petal_width": 0.2,
  "species": "setosa"
}
```

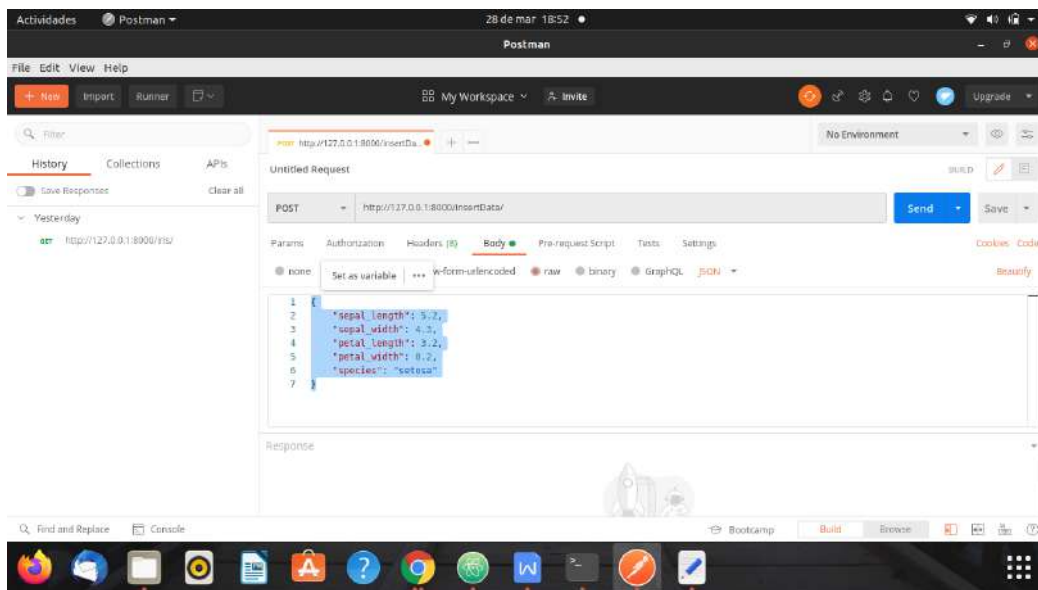


Figura 4.27: Comprobación de deserción de un dato en Postman

Damos a send:

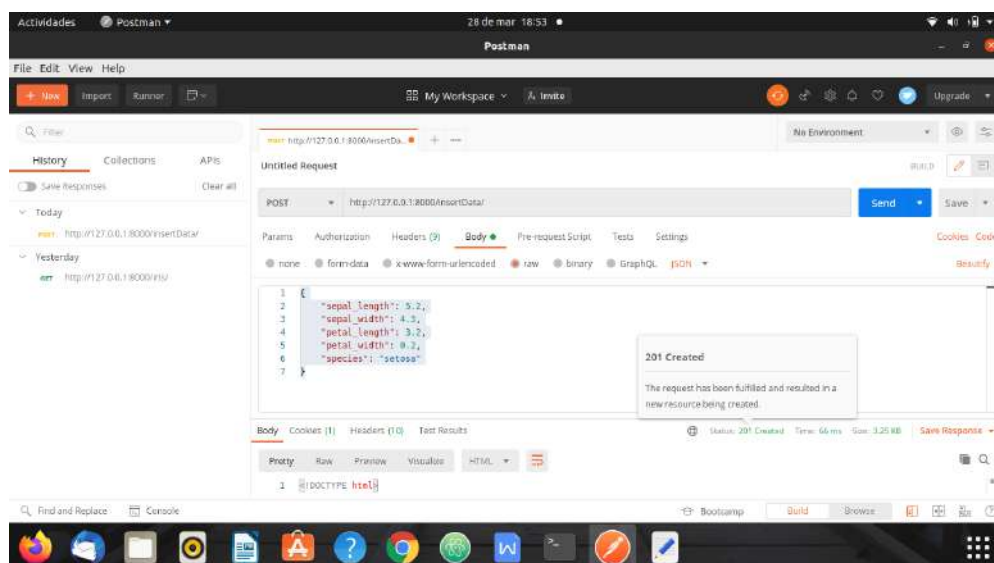
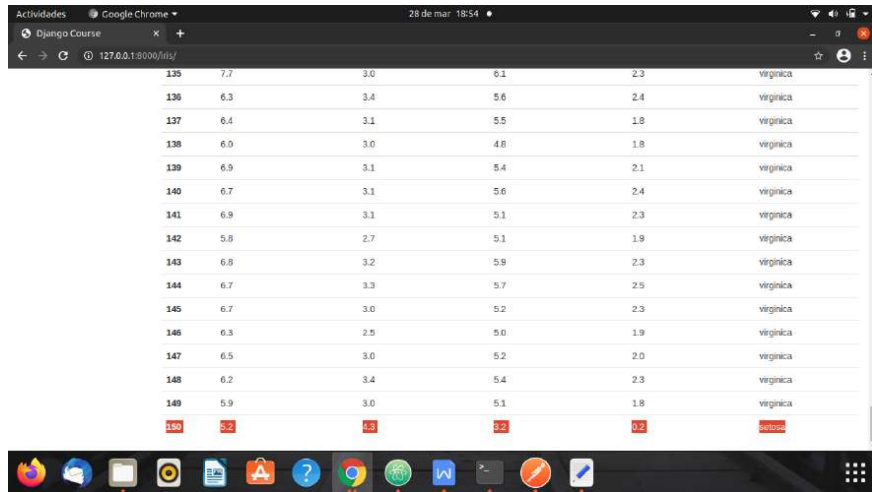


Figura 4.28: Comprobación de deserción de un dato en Postman

El resultado es 201 Creado correctamente.

Si vamos a Iris data en el menú vemos que tenemos el dato con valor 150 que acabamos de insertar:



135	7.7	3.0	6.1	2.3	virginica
136	6.3	3.4	5.6	2.4	virginica
137	6.4	3.1	5.5	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica
139	6.9	3.1	5.4	2.1	virginica
140	6.7	3.1	5.6	2.4	virginica
141	6.9	3.1	5.1	2.3	virginica
142	5.8	2.7	5.1	1.9	virginica
143	6.8	3.2	5.9	2.3	virginica
144	6.7	3.3	5.7	2.5	virginica
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica
150	5.2	4.3	3.2	0.2	setosa

Figura 4.29: Comprobación de deserción de un dato

Si mandamos los datos erróneamente , por ejemplo un dato como text y una coma:

```
{
  "sepal_length": 5.2,
  "sepal_width": 4.3,
  "petal_length": 3.2,
  "petal_width": "0,2",
  "species": "setosa"
}
```

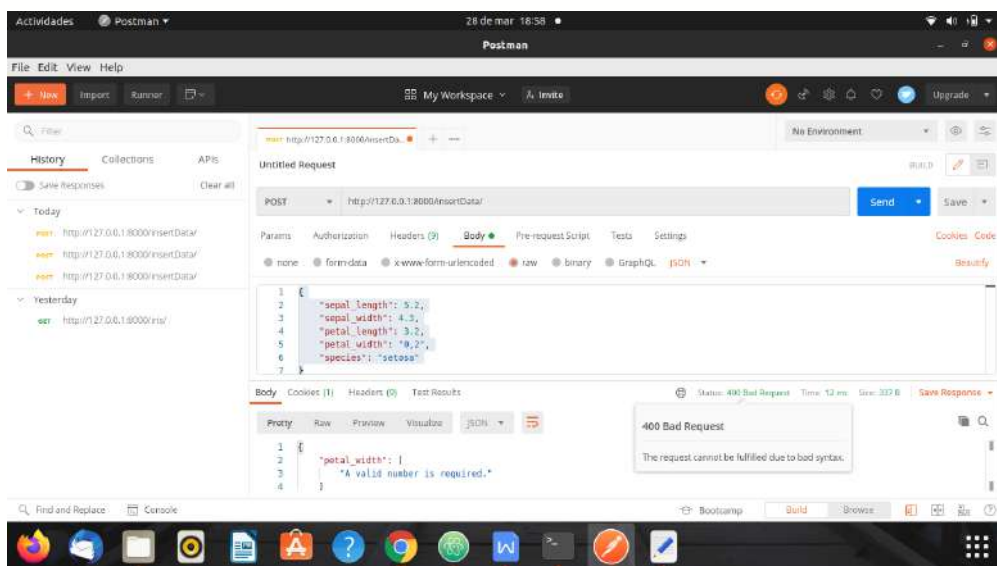


Figura 4.30: Comprobación de deserción de un dato en Postman

Nos da un error 400 y en el body nos dice el tipo de error:

```
{
  "petal_width": [
    "A valid number is required."
  ]
}
```

### 4.3 Método PUT

Creamos la plantilla para el método PUT en nuestro proyecto, como hemos comentado anteriormente mediante HTML sólo es posible emplear el POST para realizar el método PUT será necesario el POSTMAN. La plantilla para el update.html es:

```
{% extends "base_generic.html" %}
{% block content %}
<html>
  <body>
    <form method='post'>
      {% csrf_token %}
      <h1>Iris Dataset</h1>
      <br>
      <h3>· Update Data:</h3>
      <table class="table table-hover">
        <tr>
          <th><label>Sepal Length:</label></th>
          <td><input name="sepal_length" value="{{ sepal_length }}"
type="number" step='0.1' /></td>
        </tr>
        <tr>
          <th><label>Sepal Width:</label></th>
          <td><input name="sepal_width" value="{{ sepal_width }}"
type="number" step='0.1' /></td>
        </tr>
        <tr>
          <th><label>Petal Length:</label></th>
          <td><input name="petal_length" value="{{ petal_length }}"
type="number" step='0.1' /></td>
        </tr>
        <tr>
          <th><label>Petal Width:</label></th>
          <td><input name="petal_width" value="{{
lastDate.petal_width }}" type="number" step='0.1' /></td>
        </tr>
        <tr>
          <th><label>Species:</label></th>
          <td><input name="species" value="{{ species }}"
type="text" /></td>
        </tr>
      </table>
      <input type="submit" name="Submit">
    </form>
  </body>
</html>
{% endblock %}
```

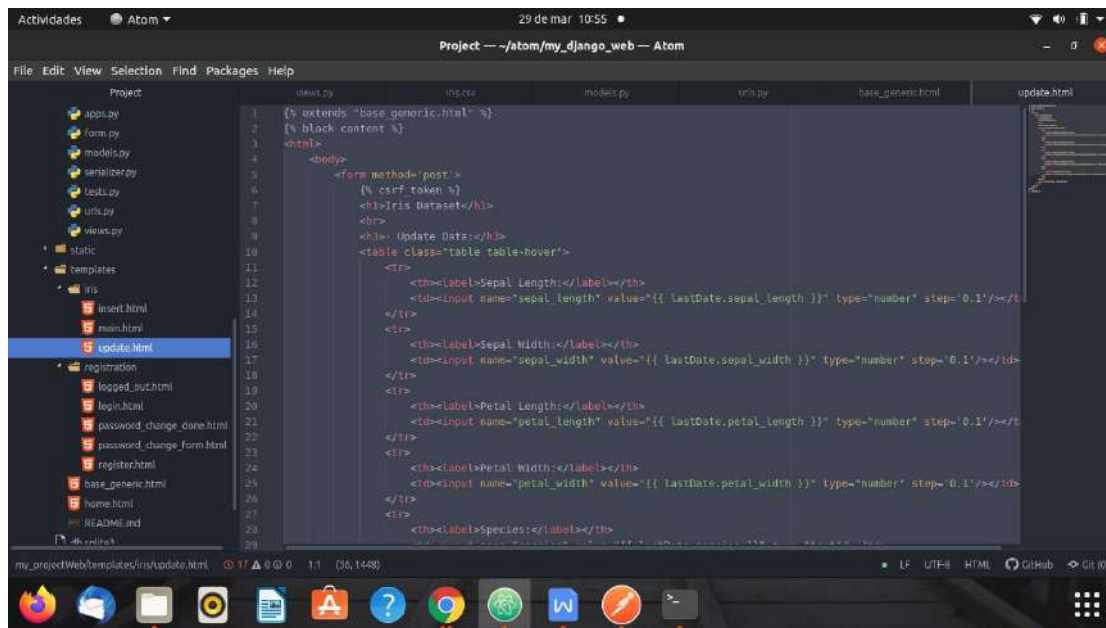


Figura 4.31: Archivo update.html

Entre paréntesis colocamos el valor que vendrá al realizar el GET como context.

Creamos la función en myApp --> iris --> views.py

Habilitamos los métodos GET, PUT, POST para la función:

```
...
from django.shortcuts import render, redirect
...
@api_view(['GET', 'PUT', 'POST'])
def updateData(request):
    if request.method == 'GET':
        # Ruta donde se encuentra nuestro archivo:
        #
        /home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv
        X = settings.MEDIA_ROOT + '/iris.csv'
        # Cargamos el dataset con ayuda de pandas:
        X_df = pd.read_csv(X)
        lastDate = X_df.iloc[-1]
        sepal_length = str(lastDate['sepal_length'])
        sepal_width = str(lastDate['sepal_width'])
        petal_length = str(lastDate['sepal_width'])
        petal_width = str(lastDate['petal_width'])
        return render(request, 'iris/update.html', context={
            'lastDate': lastDate, 'sepal_length': sepal_length,
            'sepal_width': sepal_width, 'petal_length':
            petal_length,
            'petal_width': petal_width})
        # Lo probamos usando POSTMAN:
    elif request.method == 'PUT':
        # definir 'data' como el conjunto de datos
```

```

        # que se reciben a través del front-end:
        data = request.data
        print(data)
        # Ruta donde se encuentra nuestro archivo:
        #
/home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv
        X = settings.MEDIA_ROOT + '/iris.csv'
        df = pd.read_csv(X)
        # Pasaremos los datos por el serializer
        # para comprobar si los datos siguen el modelo esperado:
        serializer = irisSerializer(data=data)
        if serializer.is_valid():
            # Guardamos los datos:
            serializer.save()
            # sustituimos la última fila del dataset cada uno de los
valores
            # para obtener el dato serializer.data:
            df.loc[df.index[-1], 'sepal_length'] =
serializer.data['sepal_length']
            df.loc[df.index[-1], 'sepal_width'] =
serializer.data['sepal_width']
            df.loc[df.index[-1], 'petal_length'] =
serializer.data['petal_length']
            df.loc[df.index[-1], 'petal_width'] =
serializer.data['petal_width']
            df.loc[df.index[-1], 'species'] = serializer.data['species']
            df.to_csv(X, index=False)
            return Response(df.iloc[-1], status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    # Lo mismo que el método PUT pero a través del front-end:
    elif request.method == 'POST':
        # definir 'data' como el conjunto de datos
        # que se reciben a través del front-end:
        data = request.data
        print(data)
        # Ruta donde se encuentra nuestro archivo:
        #
/home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv
        X = settings.MEDIA_ROOT + '/iris.csv'
        df = pd.read_csv(X)
        # Pasaremos los datos por el serializer
        # para comprobar si los datos siguen el modelo esperado:
        serializer = irisSerializer(data=data)
        if serializer.is_valid():
            # Guardamos los datos:
            serializer.save()
            # sustituimos la última fila del dataset cada uno de los
valores
            # para obtener el dato serializer.data:
            df.loc[df.index[-1], 'sepal_length'] =
serializer.data['sepal_length']
            df.loc[df.index[-1], 'sepal_width']
=serializer.data['sepal_width']

```

```

        df.loc[df.index[-1], 'petal_length'] =
serializer.data['petal_length']
        df.loc[df.index[-1], 'petal_width'] =
serializer.data['petal_width']
        df.loc[df.index[-1], 'species'] = serializer.data['species']
        # convertir a csv
        df.to_csv(X, index=False)
        # Redireccionamos a la página principal para comprobar el
dataset:
        return redirect('/iris/')
    return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)

```

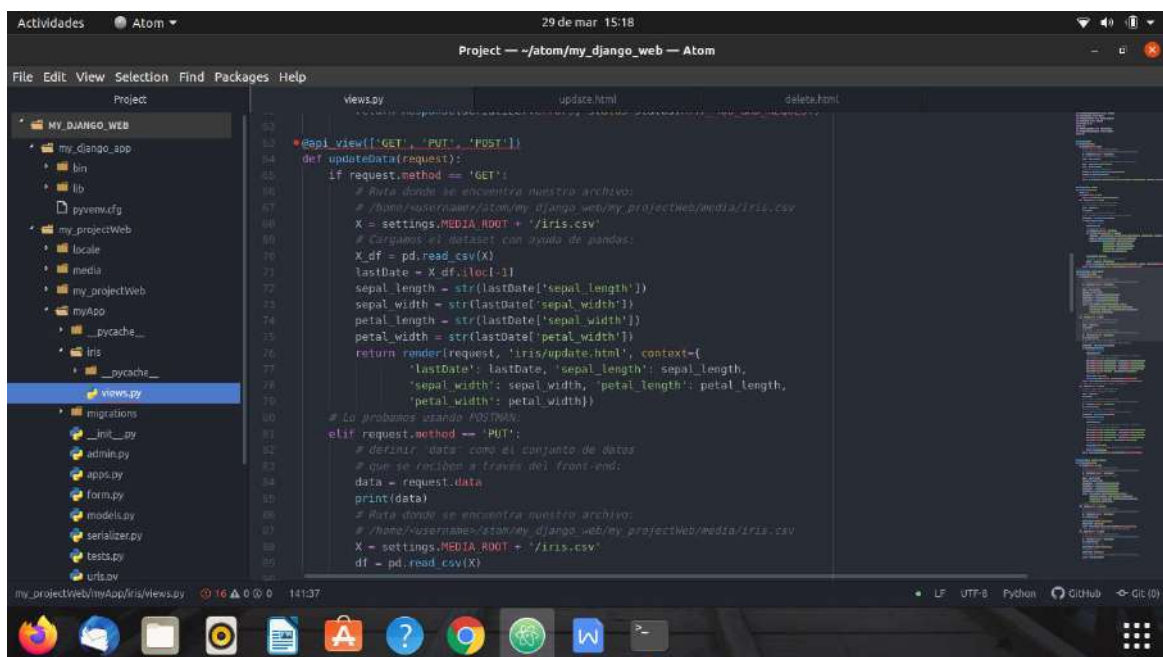


Figura 4.32: Archivo views.py

Creamos la url para este nuevo método en myApp--> urls.py:

```

from myApp.iris.views import irisData, insertData, updateData

urlpatterns = [

    ...,

    url(r"^updateData/", updateData, name="updateData"),

]

```



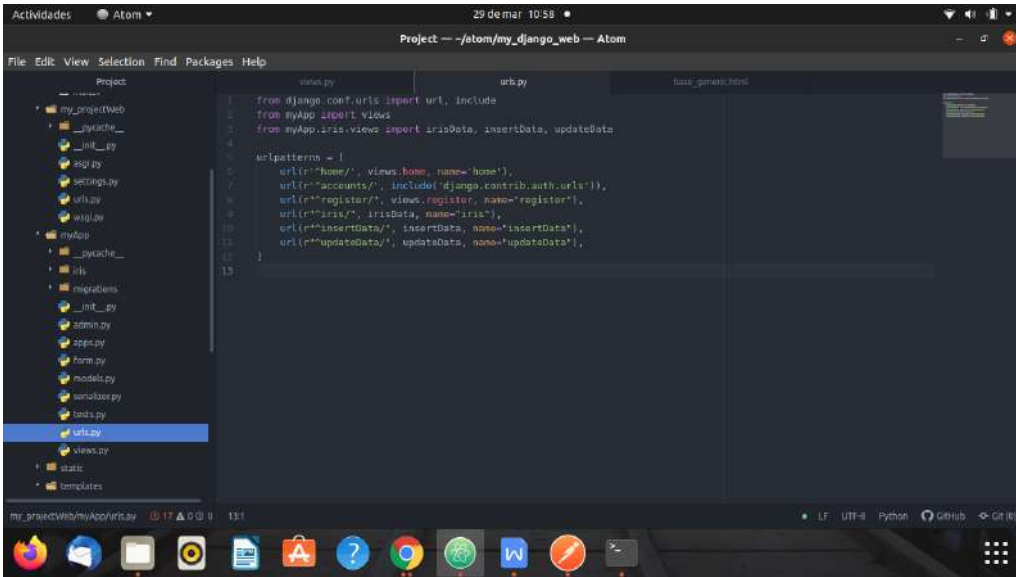


Figura 4.33: Archivo `urls.py`

En la plantilla templates--> basic\_generic.html añadimos el nuevo botón al menú:

```
<li><a href="{% url 'updateData' %}">Update Data</a></li>
```

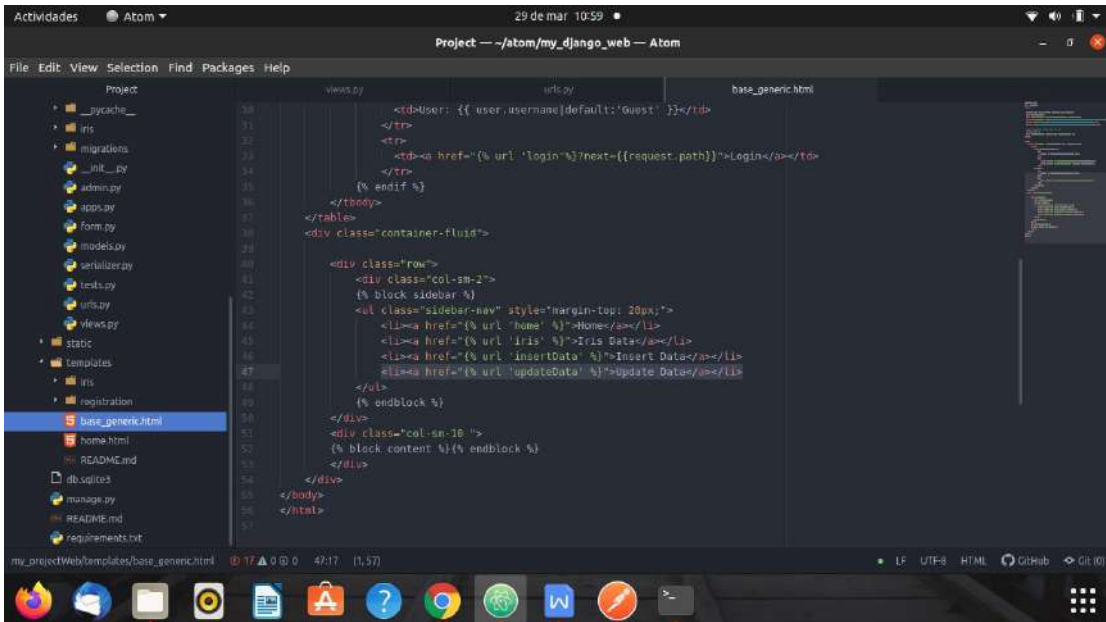


Figura 4.34: Archivo *basic\_generic.html* añadir el botón *Update Data*



Vamos a nuestro proyecto y vemos que se nos ha añadido un nuevo botón Update Data:

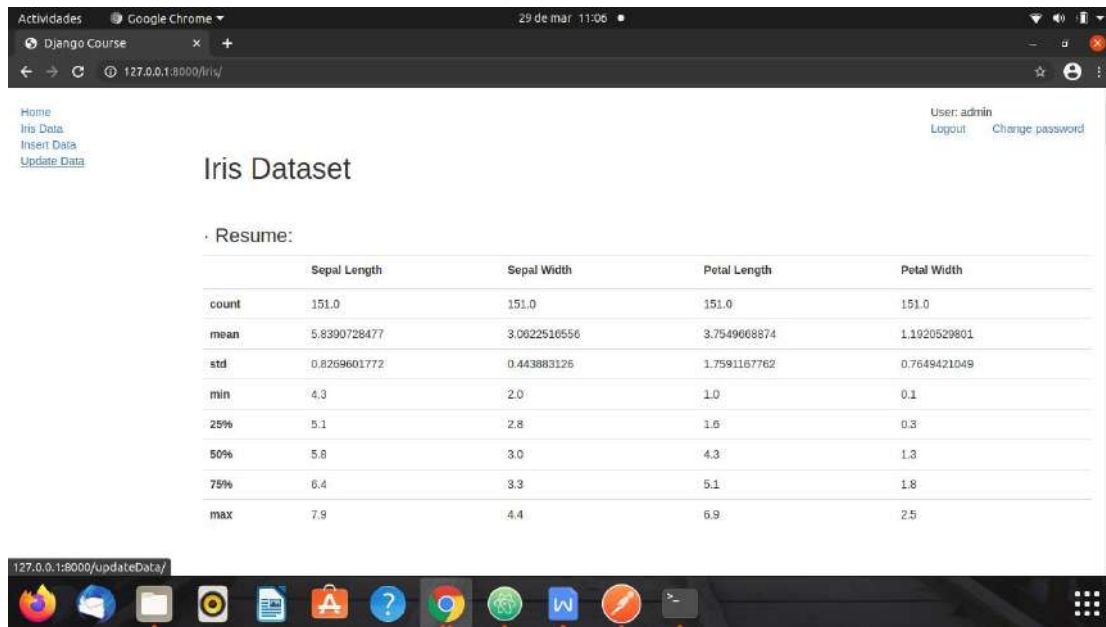


Figura 4.35: Pantalla donde se ve el botón creado Update Data en el menú de la esquina izquierda arriba.

Hacemos clic y nos aparecerá la siguiente pantalla:

Update Data:

Sepal Length:

Sepal Width:

Petal Length:

Petal Width:

Species:

Figura 4.36: Pantalla de actualización del último dato del dataset.

Vemos que nos carga los valores del último dato a modificar, modificamos algún valor y damos a enviar:

Actividades Google Chrome 29 de mar. 11:07

Django Course x +

127.0.0.1:8000/updateData/

Home  
Iris Data  
Insert Data  
Update Data

User: admin  
Logout Change password

### Iris Dataset

Update Data:

Sepal Length: 5.4

Sepal Width: 4.3

Petal Length: 3.2

Petal Width: 0.2

Species: versicolor

Enviar

Figura 4.37: Pantalla de actualización del último dato del dataset.

Una vez hacemos clic en enviar nos devuelve a la ventana de los datos y vemos que se ha actualizado correctamente:

Actividades Google Chrome 29 de mar. 11:09

Django Course x +

127.0.0.1:8000/iris/

135	7.7	3.0	6.1	2.3	virginica
136	6.3	3.4	5.6	2.4	virginica
137	6.4	3.1	5.5	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica
139	6.9	3.1	5.4	2.1	virginica
140	6.7	3.1	5.6	2.4	virginica
141	6.9	3.1	5.1	2.3	virginica
142	5.8	2.7	5.1	1.9	virginica
143	6.8	3.2	5.9	2.3	virginica
144	6.7	3.3	5.7	2.5	virginica
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica
150	5.4	4.3	3.2	0.2	versicolor

Figura 4.38: Comprobación de actualización de un dato

Ahora vamos a probar lo mismo con el método PUT en el Postman para ello ponemos la url <http://127.0.0.1:8000/updateData/>

Y ponemos método PUT en el body --> raw --> json añadimos:

```
{  
  "sepal_length": 5.4,  
  "sepal_width": 4.3,  
  "petal_length": 3.2,  
  "petal_width": 0.2,  
  "species": "versicolor"  
}
```

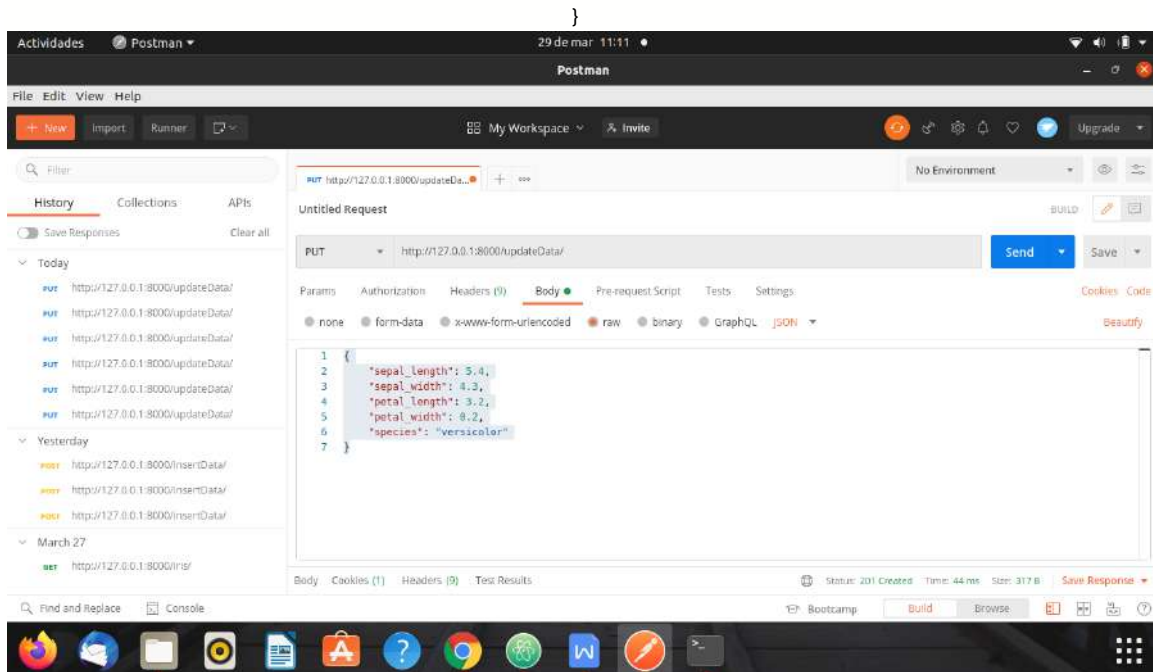


Figura 4.39: Actualización de un dato en Postman

Damos a send:

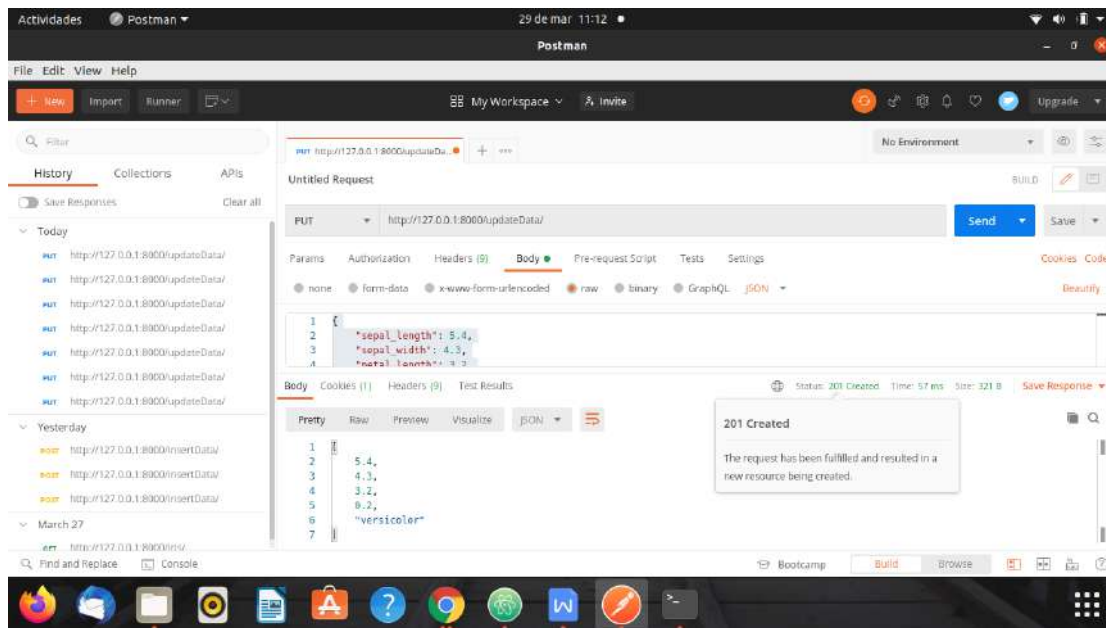


Figura 4.40: Actualización de un dato en Postman

Nos da la respuesta 201 Created y nos envía el dato actualizado:

[5.4, 4.3, 3.2, 0.2, "versicolor"]

Si vamos a nuestra aplicación de vuelta veremos que se ha actualizado el dato igual que a través del front pero empleado el método PUT:

The screenshot shows a web application displaying a table of Iris dataset records. The table has 6 columns: ID, sepal\_length, sepal\_width, petal\_length, petal\_width, and species. The last record (ID 150) is highlighted in red, showing the updated values: [5.4, 4.3, 3.2, 0.2, "versicolor"].

ID	sepal_length	sepal_width	petal_length	petal_width	species
135	7.7	3.0	6.1	2.3	virginica
136	6.3	3.4	5.6	2.4	virginica
137	6.4	3.1	5.5	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica
139	6.9	3.1	5.4	2.1	virginica
140	6.7	3.1	5.6	2.4	virginica
141	6.9	3.1	5.1	2.3	virginica
142	5.8	2.7	5.1	1.9	virginica
143	6.8	3.2	5.9	2.3	virginica
144	6.7	3.3	5.7	2.5	virginica
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica
150	5.4	4.3	3.2	0.2	versicolor

Figura 4.41: Comprobación de actualización de un dato

## 4.4 Método DELETE

Creemos la plantilla que vamos a usar para este método igual que pasa con el método DELETE NO es posible usarlo con html para ello definimos el método POST:

```
{% extends "base_generic.html" %}
{% block content %}
<html>
  <body>
    <form method='post'>
      {% csrf_token %}
      <h1>Iris Dataset</h1>
      <br>
      <h3>· Delete Data:</h3>
      <table class="table table-hover">
        <tr>
          <th><label>Sepal Length:</label></th>
          <td><input name="sepal_length" value="{{
sepal_length }}" type="number" step='0.1' readonly/></td>
        </tr>
        <tr>
          <th><label>Sepal Width:</label></th>
          <td><input name="sepal_width" value="{{ sepal_width
}}" type="number" step='0.1' readonly/></td>
        </tr>
        <tr>
          <th><label>Petal Length:</label></th>
          <td><input name="petal_length" value="{{
petal_length }}" type="number" step='0.1' readonly/></td>
        </tr>
        <tr>
          <th><label>Petal Width:</label></th>
          <td><input name="petal_width" value="{{ petal_width
}}" type="number" step='0.1' readonly/></td>
        </tr>
        <tr>
          <th><label>Species:</label></th>
          <td><input name="species" value="{{
lastDate.species }}" type="text" readonly/></td>
        </tr>
      </table>
      <input type="submit" name="Delete" value="Delete">
    </form>
  </body>
</html>
{% endblock %}
```

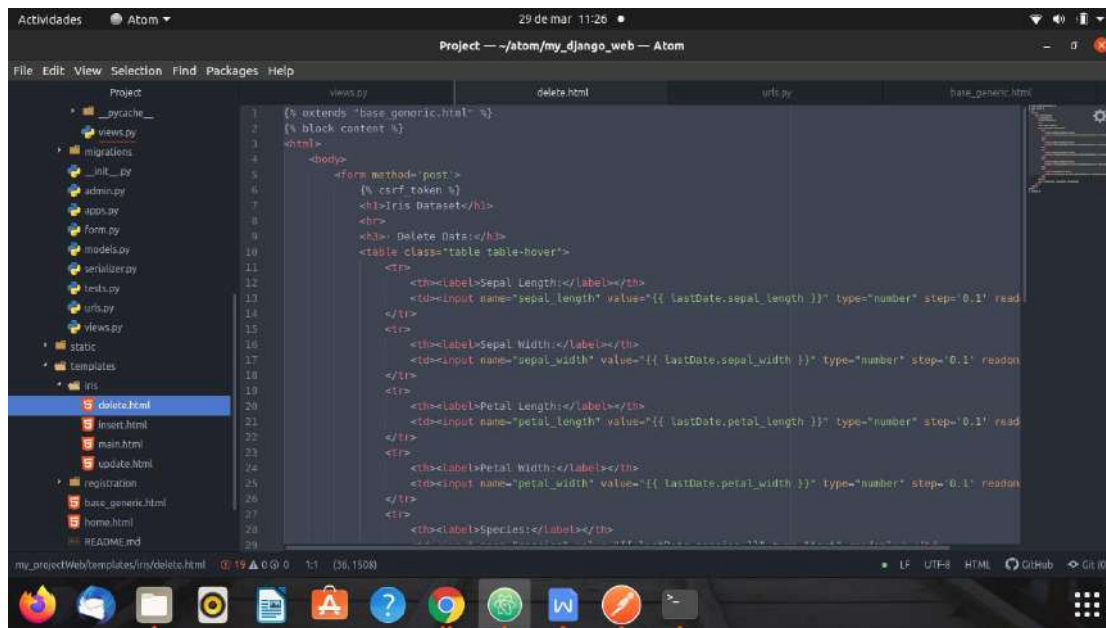


Figura 4.42: Archivo delete.html

Creamos una función nueva en myApp --> iris --> views.py:

```
@api_view(['GET', 'DELETE', 'POST'])
def deleteData(request):
    if request.method == 'GET':
        # Ruta donde se encuentra nuestro archivo:
        # /home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv
        X = settings.MEDIA_ROOT + '/iris.csv'
        # Cargamos el dataset con ayuda de pandas:
        X_df = pd.read_csv(X)
        lastDate = X_df.iloc[-1]
        sepal_length = str(lastDate['sepal_length'])
        sepal_width = str(lastDate['sepal_width'])
        petal_length = str(lastDate['sepal_width'])
        petal_width = str(lastDate['petal_width'])
        return render(request, 'iris/delete.html', context={
            'lastDate': lastDate, 'sepal_length': sepal_length,
            'sepal_width': sepal_width, 'petal_length': petal_length,
            'petal_width': petal_width})
        # Lo probamos usando POSTMAN:
    elif request.method == 'DELETE':
        # Ruta donde se encuentra nuestro archivo:
```

```
# /home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv

X = settings.MEDIA_ROOT + '/iris.csv'

df = pd.read_csv(X)

# Eliminar la última fila

df.drop(df.index[-1], inplace=True)

df.to_csv(X, index=False)

return Response(df.iloc[-1], status=status.HTTP_204_NO_CONTENT)

# Lo mismo que el método DELETE pero a través del front-end:

elif request.method == 'POST':

    # Ruta donde se encuentra nuestro archivo:

    # /home/<username>/atom/my_django_web/my_projectWeb/media/iris.csv

    X = settings.MEDIA_ROOT + '/iris.csv'

    df = pd.read_csv(X)

    # Eliminar la última fila

    df.drop(df.index[-1], inplace=True)

    # convertir a csv

    df.to_csv(X, index=False)

    # Redireccionamos a la página principal para comprobar el dataset:

    return redirect('/iris/')
```

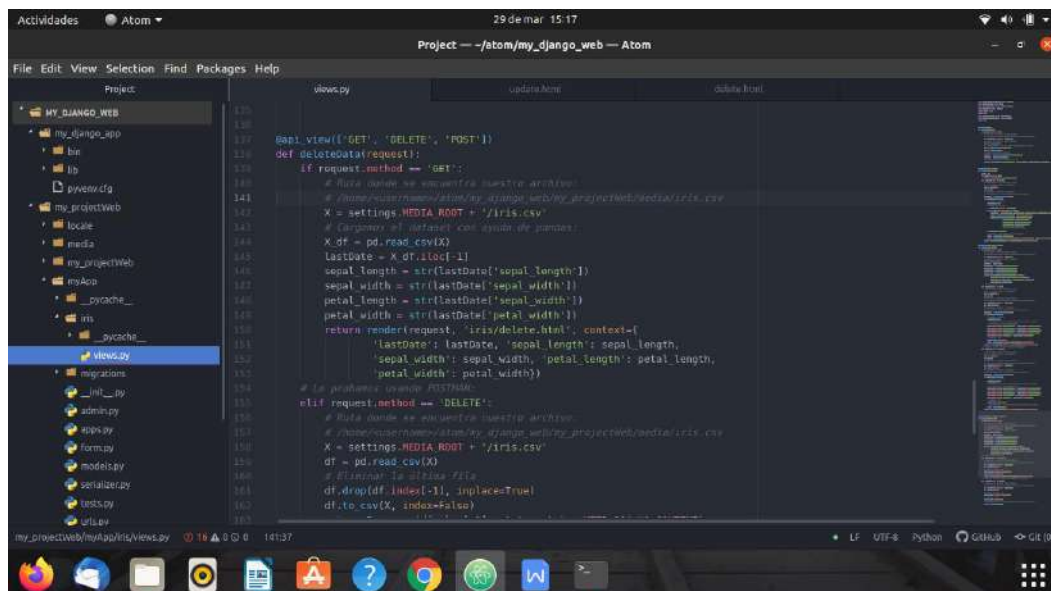


Figura 4.43: Archivo views.py



Nombramos la url en myApp --> urls.py:

```
from myApp.iris.views import irisData, insertData, updateData, deleteData

urlpatterns = [...,
    url(r"^deleteData/", deleteData, name="deleteData"),
]
```

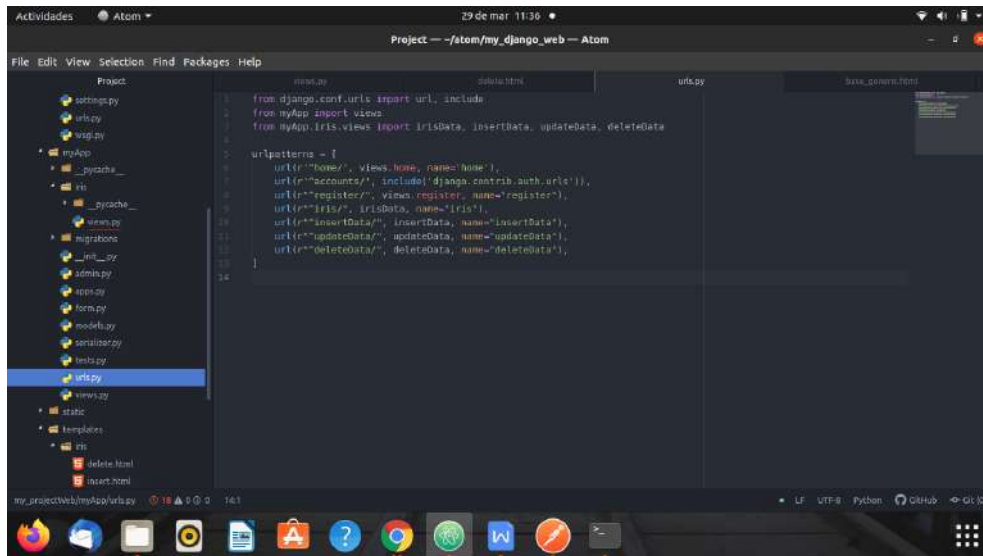


Figura 4.44: Archivo urls.py

Vamos a nuestra plantilla principal templates --> basic\_generic.html y añadimos el botón al menú:

```
<li><a href="{% url 'deleteData' %}">Delete Data</a></li>
```

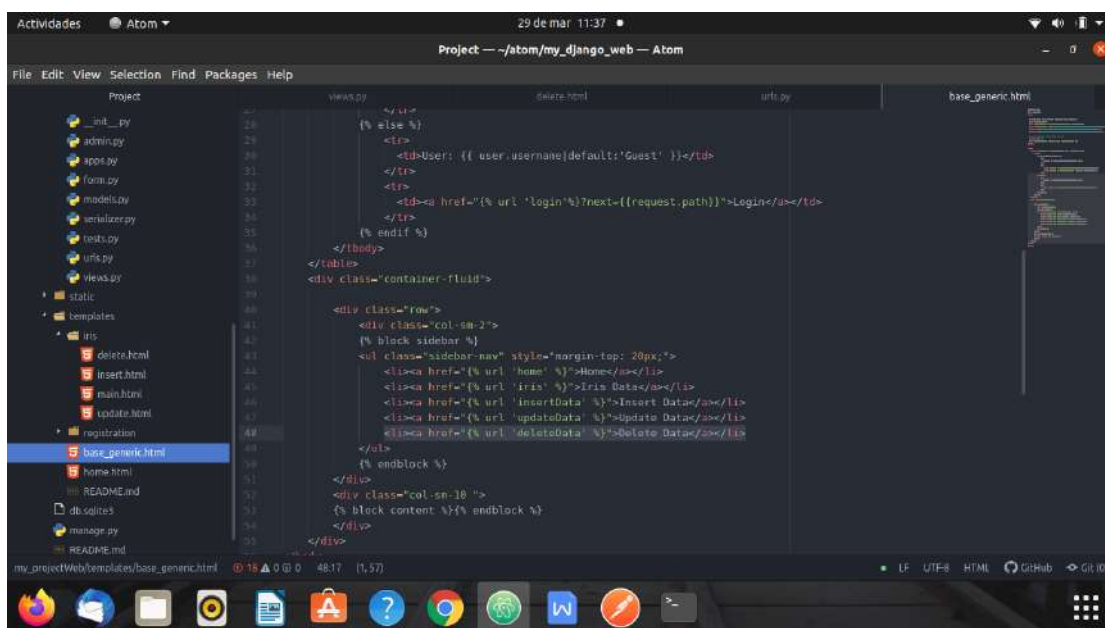


Figura 4.45: Archivo basic\_generic.html botón añadido Delete Data



Vamos al navegador y recargamos nuestra página para visualizar nuestro nuevo botón:

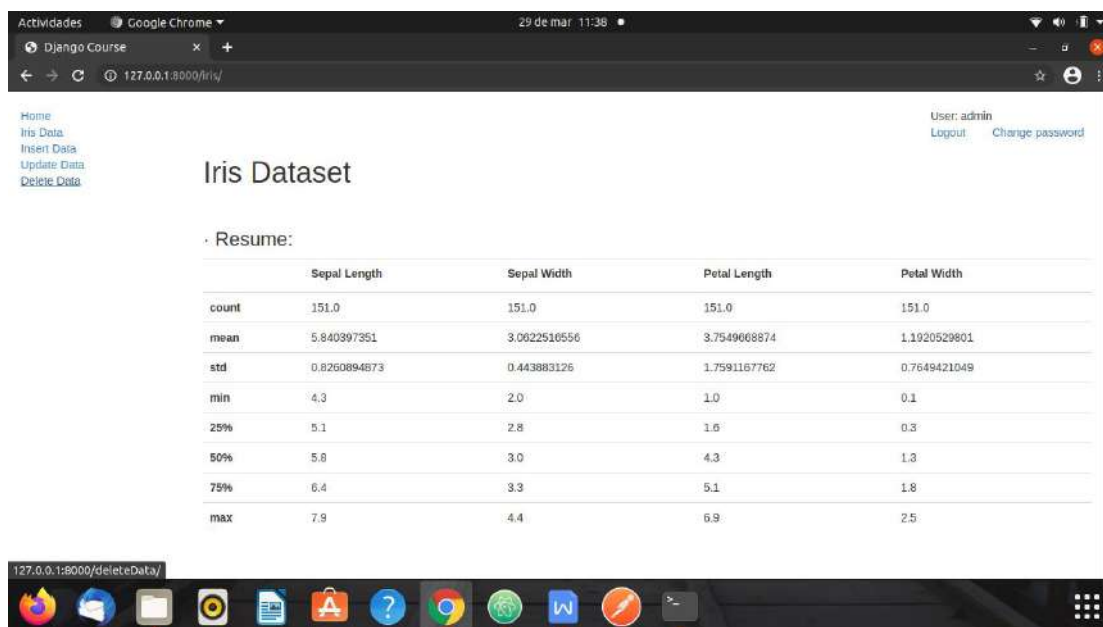


Figura 4.46: Pantalla donde se ve el botón añadido Delete Data en el menú

Hacemos clic al botón:

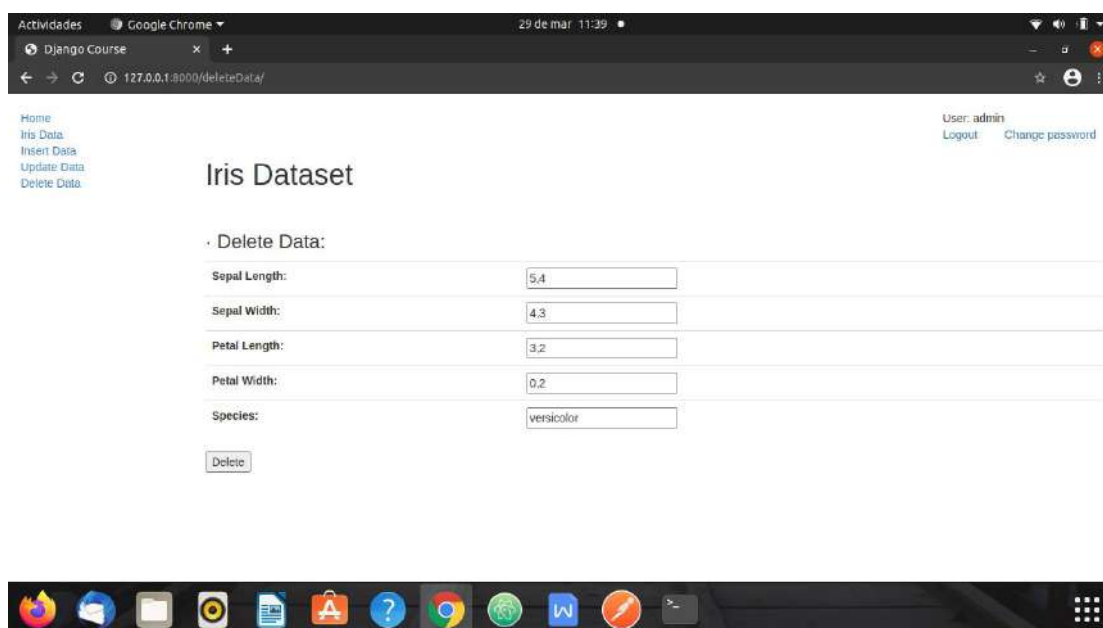
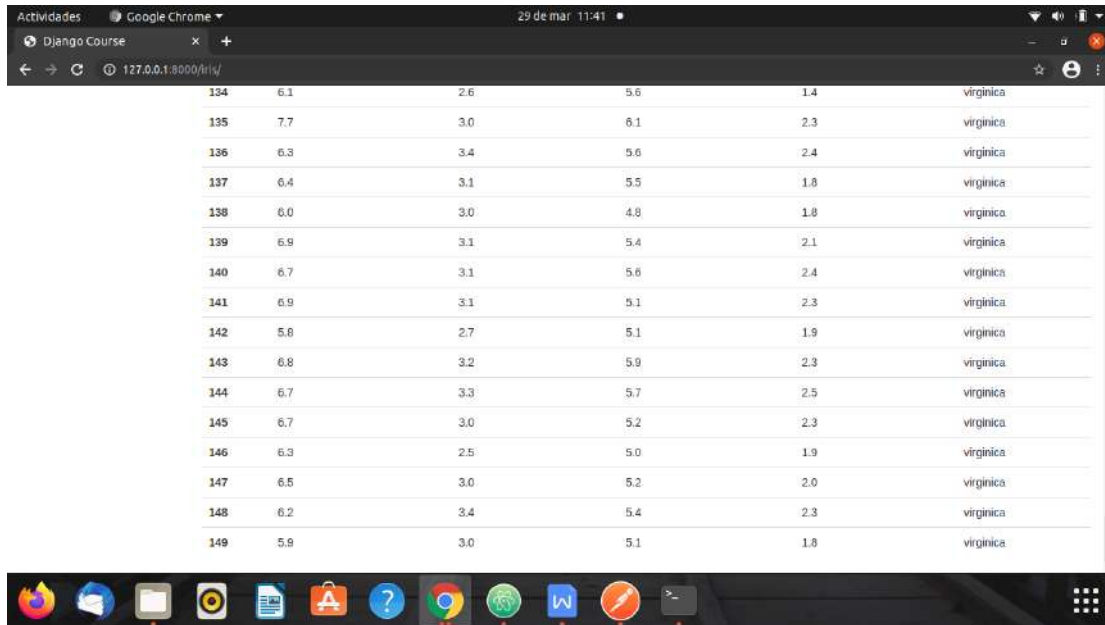


Figura 4.47: Pantalla para eliminar el último dato del csv

Nos muestra el último dato en nuestro csv que vamos a eliminar, hacemos clic en el botón de eliminar y nos retorna a la ventana del dataset y observamos que el dato se ha eliminado:



134	6.1	2.6	5.6	1.4	virginica
135	7.7	3.0	6.1	2.3	virginica
136	6.3	3.4	5.6	2.4	virginica
137	6.4	3.1	5.5	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica
139	6.9	3.1	5.4	2.1	virginica
140	6.7	3.1	5.6	2.4	virginica
141	6.9	3.1	5.1	2.3	virginica
142	5.8	2.7	5.1	1.9	virginica
143	6.8	3.2	5.9	2.3	virginica
144	6.7	3.3	5.7	2.5	virginica
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

Figura 4.48: Imagen donde vemos que se ha eliminado el último dato

Realizamos la misma operación con el Postman pero empleando el método DELETE a la url: <http://127.0.0.1:8000/deleteData/> Body --> none

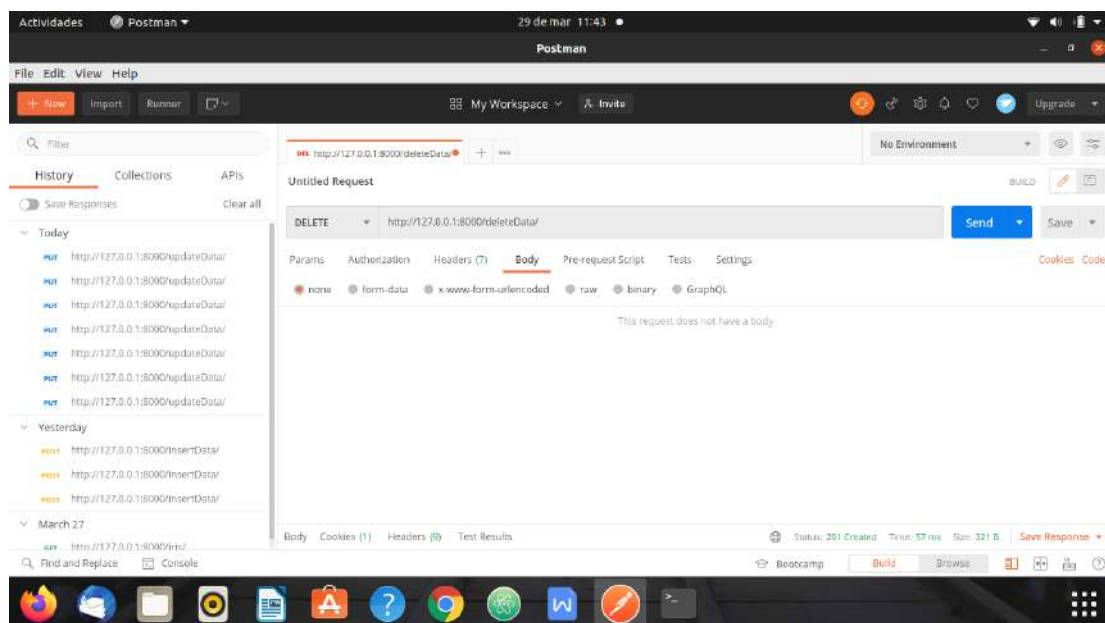


Figura 4.49: Eliminación del último dato usando Postman

Le damos a send:

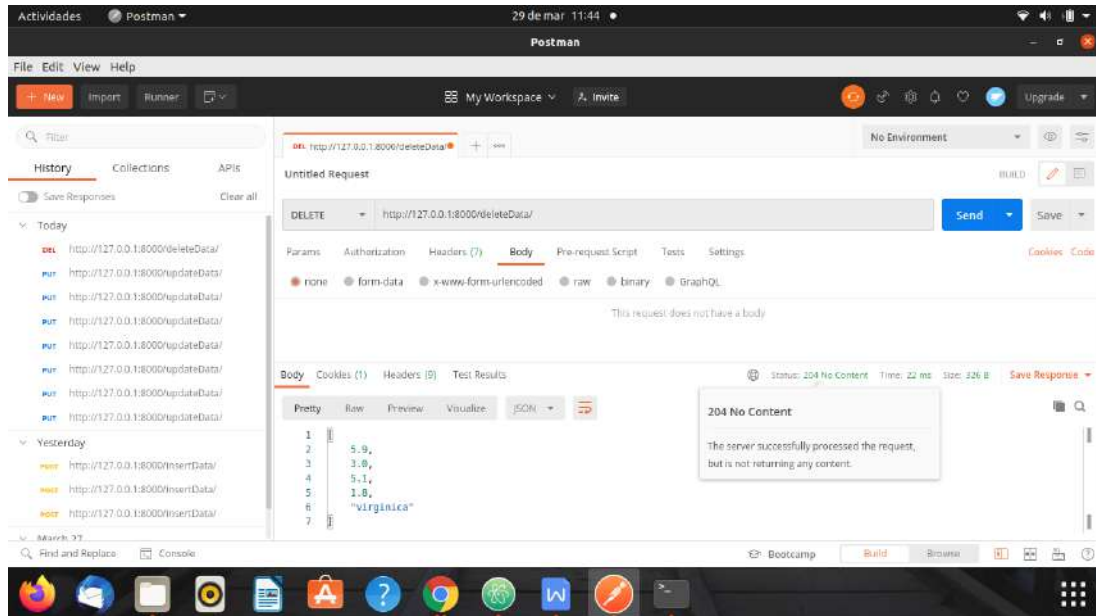


Figura 4.50: Eliminación del último dato usando Postman

Nos da un código de estado 204 No content que se nos ha eliminado correctamente y nos muestra el último valor de nuestro dataset.

## 5. PUNTOS CLAVE

Los métodos Básicos para realizar una API REST son GET (lectura), POST (Insertar), PUT (actualizar) y DELETE (eliminar).

